新工科建设之路·软件工程规划教材

软 件 工 程

朴 勇 周 勇 编著

電子工業出版社
Publishing House of Electronics Industry
北京·BEIJING

内 容 简 介

本书主要围绕软件的系统工程化开发过程，介绍相关的理论、方法、技术和工具。本书以面向对象的分析和设计为主线，以基本理论为出发点，遵循 UML 2 标准，介绍软件开发的阶段组织及生命周期模型；重点针对软件开发的具体活动，讨论需求分析、软件架构的构建、类的分析与设计、代码生成、类的详细设计、设计优化、实现技术、交互设计、软件测试等重要环节；介绍软件项目级管理及软件过程管理与改进。

本书内容丰富、循序渐进，注重软件工程理论与实践的结合，适合作为高等院校计算机相关专业本科生和研究生的教材，可也为从事软件开发的理论研究人员和工程技术人员提供参考。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

软件工程 / 朴勇，周勇编著. —北京：电子工业出版社，2019.8
ISBN 978-7-121-33854-0

I. ①软… II. ①朴… ②周… III. ①软件工程－高等学校－教材 IV. ①TP311.5

中国版本图书馆 CIP 数据核字(2018)第 049564 号

责任编辑：章海涛
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×1092　1/16　印张：17　　字数：435 千字
版　　次：2019 年 8 月第 1 版
印　　次：2019 年 8 月第 1 次印刷
定　　价：49.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：liuy01@phei.com.cn。

前　言

“软件工程”是各高校软件学院开设的一门专业基础课程，主要介绍软件工程的基本原理、开发方法和工具，是软件开发经验总结的理论课程，同时也具有很强的实践性。另外，该课程涉及计算机、经济学、管理学、工程学、市场学等多个领域的知识，具有知识广泛性的特点。

在多年教学实践中，编者深切体会到该课程授课过程中的诸多问题，比如理论知识的枯燥乏味，使学生不容易将抽象的理论与实践联系起来；没有大项目案例的依托，学生体会不到软件工程的作用等。如何突出重点、使学生掌握必备的软件工程基础及应用案例，对软件工程课程的讲解提出了挑战。

目前，软件工程领域的教材在一定程度上存在内容陈旧、偏重过程、实践及案例内容欠缺等问题。本书突出实用的特点，一方面涵盖软件工程的主要知识点，强调新技术、新方法、新理念；另一方面以案例为线索，以面向对象方法为主线，将分散的知识点连贯起来，便于读者的理解和消化，注重理论和实践的结合，为读者提供一条循序渐进的学习路线。

本书的具体内容包括：第 1 章介绍软件工程的相关概念、技术与方法；第 2 章围绕软件开发过程，对软件开发的阶段组织及生命周期模型进行说明；第 3 章到第 11 章主要围绕软件开发的具体活动，讨论需求分析、软件架构的构建、类的分析与设计、代码生成、类的详细设计、设计优化、实现技术、交互设计、软件测试等内容；第 12 章介绍软件项目级管理，包括软件配置管理、项目计划、项目计划跟踪控制、软件质量保证等；第 13 章重点介绍软件过程管理与改进。本书知识结构紧凑、突出案例教学。在开发方法的介绍上通过具体案例贯穿，并且尽量以程序代码的形式对相关的知识点进行说明和展示，以代码为驱动，以此鼓励学生通过实践，对软件工程进行深入的理解和探索。本书提供配套教学资源(PPT、源代码)，读者可登录华信教育资源网(www.hxedu.com.cn)免费下载。

本书适合作为高等院校计算机相关专业本科生和研究生的教材，可也为从事软件开发的理论研究人员和工程技术人员提供参考。

限于作者的水平及精力，书中难免有不妥之处，敬请各位读者批评指正，我们将虚心改正。

编　者

目　录

第 1 章　软件工程概述 …… 1
1.1　软件危机与软件工程 …… 1
1.1.1　软件危机 …… 1
1.1.2　软件工程 …… 2
1.2　系统工程与 UML …… 3
1.2.1　系统工程 …… 3
1.2.2　统一建模语言 UML …… 4
1.3　系统开发的解空间 …… 6
1.4　软件工程开发方法 …… 8
1.4.1　传统方法 …… 8
1.4.2　面向对象方法 …… 8
1.4.3　理解两种开发方法 …… 9
1.5　习题 …… 10

第 2 章　软件开发过程 …… 11
2.1　软件生命周期模型与软件过程 …… 11
2.2　传统生命周期模型 …… 13
2.2.1　瀑布模型 …… 13
2.2.2　快速原型模型 …… 14
2.2.3　增量模型 …… 16
2.2.4　螺旋模型 …… 16
2.2.5　喷泉模型 …… 18
2.3　敏捷生命周期模型 …… 18
2.3.1　增量交付与迭代开发 …… 19
2.3.2　敏捷生命周期模型的优势 …… 20
2.3.3　极限编程 …… 21
2.3.4　Scrum …… 22
2.3.5　DevOps 过程 …… 23
2.4　习题 …… 25

第 3 章　需求分析 …… 26
3.1　需求分析活动 …… 26
3.1.1　需求准备 …… 26
3.1.2　系统涉众 …… 27
3.1.3　系统目标 …… 29
3.2　用例与系统功能 …… 29
3.2.1　用例及其表示 …… 30
3.2.2　识别用例 …… 31
3.2.3　用例规约 …… 33
3.2.4　用例提炼 …… 34
3.3　过程建模与事件流 …… 36
3.3.1　过程建模 …… 36
3.3.2　事件流 …… 40
3.4　功能性需求 …… 41
3.5　非功能性需求 …… 44
3.6　需求跟踪 …… 46
3.7　习题 …… 48

第 4 章　软件架构的构建 …… 50
4.1　软件架构及其定义 …… 50
4.1.1　软件架构的理解 …… 50
4.1.2　软件架构的定义 …… 51
4.1.3　软件架构的“4+1”视图模型 …… 51
4.2　软件架构模型 …… 53
4.3　软件架构风格 …… 53
4.3.1　管道与过滤器 …… 53
4.3.2　层次结构 …… 54
4.3.3　仓库/黑板系统 …… 55
4.3.4　正交软件结构 …… 56
4.3.5　客户机/服务器结构 …… 56
4.3.6　浏览器/服务器结构 …… 57
4.3.7　MVC 结构 …… 58
4.4　软件架构设计 …… 59
4.4.1　包及其结构 …… 59
4.4.2　包结构设计 …… 60
4.5　习题 …… 63

第 5 章 类的分析与设计 …… 64
5.1 基本类的确定 …… 64
5.1.1 类的识别 …… 65
5.1.2 初始类图 …… 66
5.1.3 类的方法 …… 67
5.1.4 类的关系 …… 68
5.1.5 类与对象 …… 70
5.2 类的细化 …… 70
5.2.1 管理类和控制类 …… 70
5.2.2 设计优化 …… 73
5.3 补充和确认 …… 74
5.3.1 顺序图 …… 74
5.3.2 场景模拟 …… 76
5.4 界面类设计 …… 80
5.5 习题 …… 82
第 6 章 代码生成 …… 83
6.1 逆向工程与 CASE 工具 …… 83
6.2 单个类的实现 …… 84
6.3 关联关系的实现 …… 86
6.4 对象归属关系的实现 …… 90
6.4.1 聚合关系 …… 90
6.4.2 组合关系 …… 92
6.4.3 依赖关系 …… 94
6.5 软件架构的实现 …… 94
6.6 其他物理实现 …… 99
6.7 习题 …… 101
第 7 章 类的详细设计 …… 103
7.1 详细设计的主要活动 …… 103
7.2 类方法的详细设计 …… 104
7.2.1 图形工具 …… 104
7.2.2 表格工具 …… 106
7.2.3 语言工具 …… 108
7.3 类的行为设计 …… 108
7.3.1 状态图的结构 …… 109
7.3.2 状态图的扩展 …… 111
7.3.3 状态图的应用 …… 112
7.4 对象约束语言 …… 114
7.5 习题 …… 118
第 8 章 设计优化 …… 121
8.1 小即是美 …… 121
8.2 设计优化思想 …… 122
8.2.1 运行时的多态 …… 122
8.2.2 耦合的消息链 …… 124
8.2.3 狎昵关系 …… 125
8.2.4 被拒绝的遗赠 …… 126
8.2.5 循环依赖 …… 126
8.3 设计原则 …… 127
8.3.1 接口隔离原则 …… 127
8.3.2 依赖倒置原则 …… 128
8.3.3 开放封闭原则 …… 128
8.3.4 Liskov 替换原则 …… 130
8.3.5 单一职责原则 …… 131
8.3.5 合成/聚合复用原则 …… 132
8.4 设计模式 …… 133
8.4.1 抽象工厂模式 …… 134
8.4.2 单例模式 …… 136
8.4.3 适配器模式 …… 138
8.4.4 桥模式 …… 139
8.4.5 装饰模式 …… 140
8.4.6 门面模式 …… 141
8.4.7 代理模式 …… 142
8.4.8 观察者模式 …… 143
8.4.9 策略模式 …… 144
8.4.10 状态模式 …… 144
8.5 习题 …… 147
第 9 章 实现技术 …… 149
9.1 非功能性需求的实现 …… 149
9.2 分布式技术 …… 151
9.3 可扩展标记语言 XML …… 153
9.4 程序轮子 …… 154
9.4.1 组件 …… 155
9.4.2 框架 …… 158
9.5 数据的持久化 …… 162
9.5.1 文件持久化 …… 162
9.5.2 数据库持久化 …… 163
9.6 领域特定语言 …… 164

9.7 模型驱动架构 …… 166
9.7.1 原理及过程 …… 166
9.7.2 MDA 的应用 …… 168
9.8 重构 …… 170
9.9 习题 …… 172
第 10 章 交互设计 …… 173
10.1 交互设计概述 …… 173
10.2 可用性 …… 174
10.2.1 任务适合性 …… 175
10.2.2 自我描述性 …… 176
10.2.3 可控性 …… 177
10.2.4 与用户期望一致性 …… 177
10.2.5 容错性 …… 179
10.2.6 可定制性 …… 179
10.2.7 易学性 …… 179
10.3 交互设计过程 …… 180
10.4 可用性的验证 …… 181
10.5 习题 …… 183
第 11 章 软件测试 …… 184
11.1 形式化验证 …… 184
11.2 测试技术 …… 186
11.2.1 测试分类 …… 186
11.2.2 测试策略 …… 187
11.2.3 非功能性测试 …… 189
11.3 软件度量 …… 189
11.3.1 控制流图与环形复杂度 …… 191
11.3.2 方法内聚缺乏度 …… 193
11.4 等价类测试 …… 194
11.4.1 等价类方法 …… 194
11.4.2 等价类与边界 …… 196
11.4.3 等价类组合 …… 197
11.4.4 面向对象中的等价类 …… 198
11.5 基于控制流的测试 …… 200
11.6 断言机制 …… 204
11.7 测试框架 …… 206
11.7.1 测试环境 …… 206
11.7.2 JUnit …… 207
11.8 可测试性 …… 213
11.9 建设性质量保证 …… 217
11.10 人工测试 …… 218
11.11 习题 …… 220
第 12 章 软件项目级管理 …… 223
12.1 软件配置管理 …… 223
12.1.1 版本管理 …… 224
12.1.2 构建管理 …… 228
12.2 项目管理 …… 230
12.2.1 项目计划与工作分解 …… 231
12.2.2 软件规模估算 …… 232
12.2.3 开发成本估算 …… 234
12.2.4 任务安排与工程网络图 …… 237
12.2.5 项目组织与甘特图 …… 238
12.3 项目计划跟踪控制 …… 239
12.3.1 项目计划跟踪 …… 239
12.3.2 挣值分析 …… 240
12.3.3 项目偏差控制 …… 242
12.4 软件质量保证 …… 243
12.4.1 质量管理 …… 243
12.4.2 软件质量保证的内容 …… 244
12.4.3 软件质量保证的任务 …… 245
12.4.4 软件质量保证计划 …… 245
12.5 风险管理 …… 245
12.6 项目人员构成与沟通 …… 247
12.6.1 项目人员构成 …… 247
12.6.2 项目人员沟通 …… 249
12.7 习题 …… 250
第 13 章 软件过程管理与改进 …… 252
13.1 软件过程管理 …… 252
13.1.1 过程改进 …… 252
13.1.2 能力成熟度模型 …… 253
13.1.3 CMMI 过程域 …… 255
13.2 个体软件过程 …… 262
13.3 团队软件过程 …… 262
13.4 习题 …… 263

第 1 章　软件工程概述

软件的开发与管理过程其实是一个优化问题，人们总是希望能在有限的资源条件下(例如，有限的预算、不断压缩的交付时限、软件工程师的数量及能力、各种风险的预期等)做到收益的最大化。在保证产品质量的前提下尽量降低软件开发的成本，软件工程的引入就是力求达到这个目的。软件工程追求软件开发的精益化，其本质是规避浪费，确保在合适的时间去做合适的事情，达到效率的最大化，这是精益思想在软件工程中的体现。

软件工程是研究和应用如何以系统性、规范化、可定量的过程化方法去开发和维护软件，以及如何把经过时间考验且被证明正确的管理技术和当前能够获得的最好的开发技术方法结合起来的学科。

本章主要介绍软件危机的产生及软件工程的由来、软件工程包括的主要内容、软件工程开发的主要方法及技术。

1.1　软件危机与软件工程

1.1.1　软件危机

软件工程的提出始于软件危机的出现。1968 年，北大西洋公约组织(NATO)在当时的联邦德国召开的国际学术会议上提出了软件危机一词，并同时提出软件工程的概念，以解决软件危机。软件危机是指在软件开发及维护的过程中遇到的一系列严重问题，这些问题可能导致软件产品的寿命缩短，甚至夭折。

软件危机在 20 世纪 70 年代表现得尤其严重，具体表现有：超预算、超期限、质量差、用户不满意、开发过程无法有效介入和管理、代码难以维护等。人们逐渐认识到软件开发是一项高难度、高风险的活动，因为它失败的可能性较大。软件危机的产生与软件本身的特点有很大的关系，其中最主要的就是软件的复杂性。

(1)软件是逻辑层面上的，不是有形的物理文件，与硬件具有完全不同的特征。而且，软件的主要成本产生于设计与研制的过程，而不是制造的环节，因为软件的制造过程可以理解为“复制”。

(2)软件在使用过程中不会磨损，但会退化。因此，软件的维护不能像维修硬件那样(进行简单的更换)。软件维护就是修复不断发现的缺陷。这个过程比较复杂，有时需要经历新的开发过程，而且缺陷被发现得越晚，为之付出的代价就越高。

(3)软件开发早期是一门艺术，但目前越来越趋于标准化，软件产业正向大规模制造和基于构件的方向前进。

(4)软件同时也是一种逻辑实体，具有抽象性。软件可以被使用，但无法看到其本身的形态。软件产品是人类智慧的作品。

(5)软件是复杂的，并且会越来越复杂。人类思想的复杂性导致了软件的复杂性。随着信息领域的发展，软件的规模会越来越大，也会越来越复杂。

人们对软件往往有着过高的期望，认为软件无所不能，对软件的认识也比较模糊。例如，早期人们对软件的误解之一就是软件即程序，软件开发就是编写程序，编写程序就是软件开发的全部工作。实际上，软件是由三部分组成的，即程序、数据和文档。程序是指能够运行的、能提供所希望的功能和性能的指令集。数据是指支持程序运行的数据。文档是指描述程序研制的过程、方法及使用的记录。随着对软件了解的深入，人们也认识到软件开发的一般性规律——变化，变化是软件开发不变的主题，变化带来了诸多挑战。

(1)软件开发各环节对缺陷具有放大作用，一个小的问题如果不及时识别和处理，经过几级放大后，会在后期带来“可观”的成本上的损失。

(2)只有早期发现问题，才会尽量减少损失，但一个难以解决的问题是，用户的需求不可能一次确定下来；甚至有时候用户对软件的需求也是模糊的，他们本身也需要一个不断学习的过程。

总之，软件危机的产生主要是由于软件的复杂性、过高的期望及无处不在的变化导致的。人们逐渐认识到应对软件危机的必要性，寻找解决软件危机的途径，主要包括以下内容。

(1)要对软件有正确的认识。

(2)推广使用软件开发成功的技术和方法，研究探索更有效的技术和方法。

(3)开发和使用更好的软件工具。

(4)对于时间、人员、资源等，需要引入更加合理的管理措施。

1.1.2 软件工程

软件工程是从技术和管理两个方面开发和维护计算机软件的一门学科。IEEE 对软件工程的定义是：将系统化、规范化、可量化的工程原则和方法应用于软件的开发、运行和维护及对其中方法的理论研究，其主要目标是高效开发高质量的软件，降低开发成本。

1999 年 5 月，ISO(国际标准化组织)和 IEC(国际电工委员会)的第一联合技术委员会启动了标准化项目——软件工程知识体系指南(Guide to the Software Engineering Body of Knowledge，SWEBOK)。SWEBOK 指南的目的是为软件工程学科的范围提供一致的确认。软件工程知识体系包含两个部分，10 个主要的知识域，如图 1.1 所示。

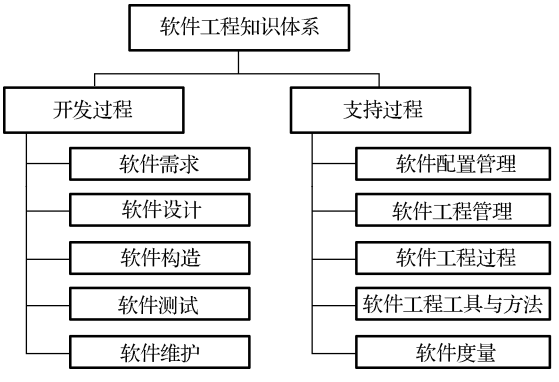


图 1.1　软件工程知识体系

软件需求(Software Requirement)是对业务领域中需要的特征的理解，包括软件需求基础、需求过程、需求获取、需求分析、需求规格说明、需求确认和实践考虑。

软件设计(Software Design)是定义系统或组件的体系结构、组成、接口和其他特征的过程，包括软件设计基础、软件设计关键问题、软件结构与体系结构、设计质量的分析与评价、软件设计表示、软件设计的策略与方法。

软件构造(Software Construction)是通过编码、验证、测试和排错的组合，具体创建一个可以工作的、有意义的软件的过程，包括软件构造基础、管理构造和实践考虑。

软件测试(Software Testing)是在有限测试用例集合上，根据期望的行为对程序的行为进行动态验证的过程，包括软件测试基础、测试级别、测试技术、测试需求、与测试相关的度量、测试过程等。

软件维护(Software Maintenance)，包括软件维护基础、软件维护的关键问题、维护过程、维护技术。软件一旦投入运行，就可能出现异常，运行环境可能发生改变，用户会提出新的需求。软件生命周期中的软件维护阶段从软件交付时开始，但是实际的维护活动出现得比其还要早。

软件配置管理(Software Configuration Management)是为了系统地控制配置的变更和维护在整个系统生命周期中的完整性和可追踪性，而标志软件在时间上不同点的配置的技术，包括软件配置过程管理、软件配置标志、软件配置控制、软件配置状态统计、软件配置审核、软件发行管理和交付。

软件工程管理(Software Engineering Management)，包括启动和范围定义、软件项目计划、软件项目实施、评审与评价、软件工程度量。

软件工程过程(Software Engineering Process)涉及软件工程过程本身的定义、实现、评估、度量、管理、变更和改进。软件工程过程包括过程实施与改变、过程定义、过程评估、过程和产品度量。

软件工程工具与方法(Software Engineering Tool and Method)，包括软件工程工具、软件工程方法。

软件质量(Software Quality)，包括软件质量基础、软件质量过程和实践考虑，是在处理跨越软件生命周期过程时的软件质量的考虑。软件质量在软件工程中无处不在，其他知识域中也涉及质量问题。

作为开发与维护的指导，软件工程的基本原理包括：用分阶段的生命周期计划严格管理；坚持进行阶段评审；实行严格的产品控制；采用现代程序设计技术；结果应能清楚地审查；开发小组的人员应该少而精；承认不断改进软件工程实践的必要性。

1.2 系统工程与 UML

1.2.1 系统工程

系统工程是为了更好地达到系统目标，对系统的构成要素、组织结构、信息流动和控制机构等进行分析与设计的技术。针对不同的领域，系统工程有着不同的实现方法，如商业过程

工程(Business Process Engineering)、产品工程(Product Engineering)等。系统工程的目的是使人们确保在正确的时间使用正确的方法做正确的事情。

系统工程用定量和定性相结合的系统思想和方法处理大型复杂系统的问题。根据一系列相关元素的合理组织,系统能够完成既定的任务或目标,如构成计算机系统的元素可以是硬件、软件、人员等,软件子系统又可以是程序、数据、文档等。所以,系统很自然的一种构成方式就是层次方式。系统分析的常用方法也是层次分析方法,它将问题分解为不同的组成因素,并按照因素间的相互关联影响及隶属关系将因素按不同层次聚集组合,形成一个多层次的分析结构模型。例如,产品工程的层次结构模型如图 1.2 所示。

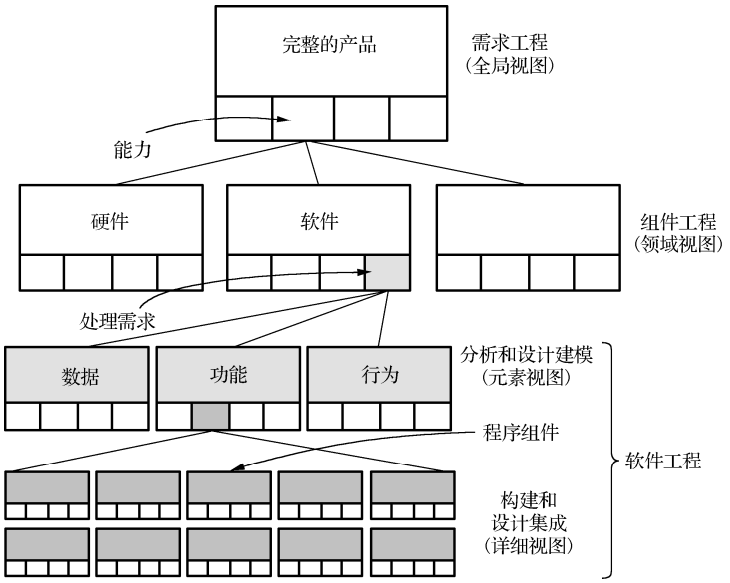


图 1.2　产品工程的层次结构模型

模型的最高层表示领域目标,即层次分析要达到的总目标;中间层表示采取某一方案来实现预定总目标所涉及的中间环节;最底层表示要选用的解决问题的各种措施、策略、方案等。

模型能更好地体现出人们对系统的理解和驾驭能力。由于系统工程本身的层次特点,系统模型本质上也是层次化的。对应系统工程的不同层次,相应的模型会被创建和使用,如需求工程中的模型要能体现出对系统宏观上的理解,下层模型要能明确具体子系统的需求。随着对系统的理解,工程化的规范会被引入,产生更细致的模型。

统一建模语言(UML)提供了一整套对系统建模的基础设施,包括模型的表示及建模的方法等,可以适用不同的系统层次。

1.2.2　统一建模语言

统一建模语言(Unified Model Language,UML)是继 20 世纪 80 年代末、90 年代初面向对象分析与设计方法(OOA&D)出现后,面向对象领域的又一个研究与讨论的热点。UML 方法统一了 Booch、OMT 及 OOSE 方法,并在此基础上进行了标准化,如今已经成为对象管理组织(Object Management Group,OMG)的标准之一。

顾名思义,UML 是一种语言,或者说是一种工具,而不是一种方法。UML 致力于分析设计

的描述，其表述形式以图形方式为主，但其描述形式本质上仍归为非正式(Informal)的方式，这区别于其他一些形式化的正式描述方法。事实上，UML 是对 OOA&D 方法的分析设计结果的展现。

历史上，三位在软件工程领域有着杰出贡献的人对 UML 的发展起着主要的推动作用，他们是 Grady Booch、James Rumbaugh 和 Ivar Jcobson，这三个人被誉为“三朋友”(The Three Amigos)。Grady Booch 是面向对象方法的最早倡导者之一，提出了面向对象软件工程的概念，他在早期与 Rational 软件公司研发 Ada 系统时做了大量工作，并由此提出了适合系统设计与构造的 Booch 方法。James Rumbaugh 一开始在 General Electric 公司曾经有一个研发团队，他们使用一种对象建模技术(OMT)，这种方法中的表示符号，独立于语言和模型并贯穿软件开发的各个阶段，实现了阶段间的平滑过渡，适用于以数据为中心的信息系统。Ivar Jacobson 早期在 Objectory 公司任职，在很多实际项目中积累了丰富的经验，提出了“用例(Use Case)”的概念，进而提出了 OOSE 方法。OOSE 以用例为中心，进行系统需求的获取、分析及高层设计等开发活动，适合支持商业工程的需求分析。

在 20 世纪 90 年代中期，以上三位分别代表着不同的学派，各自的理论也比较完善，但各有优缺点。其实，当时还有很多其他学派，如 Shlaer、Mellor、Coad 及 Yourdon 等，他们采用不同的符号进行类与对象的表示及关联，甚至出现了相似符号在不同模型中表示的意义不尽相同的现象，分别自成体系，造成了混淆，不利于大规模的软件开发活动。面对几十种不同的建模语言，这些不同的面向对象表示方法及相关的方法论形成了一个群雄争霸的时代，历史上称为“方法的战争(Method Wars)”。

终于在 1995 年，Booch 与 Rumbaugh 合作，他们将其方法合并为统一方法，并公开发表了 0.8 版本，随后他们又联合 Jacobson，在方法中加入了用例思想。1996 年，三人共同将他们的新方法命名为 UML 0.9。

但“方法的战争”并没有就此结束。1997 年 1 月，不同的组织和机构都向 OMG 提交了各自有关模型交换的草案，主要针对元模型与一些可选的表示。三人所在的 Rational 公司也将此时的 UML 1.0 版本提交给了 OMG，经过一段时间的工作和修改，取长补短，最终 OMG 选择了 UML 1.1 版本作为 OMG 的标准。从此，UML 又经过不断的修订，UML 1.3 在 1999 年成为了当时的官方版本，其也是在 UML 历史上最有意义的里程碑式的一个版本。2005 年 7 月，OMG 发布了 UML 2.0 版本，对 UML 1.x 版本进行了更新和扩充。目前，UML 也成为了 ISO 的标准之一。这些都决定了 UML 在软件开发建模领域中的地位。

UML 将软件开发中的语言表示与过程进行了分离，具有如下重要的功能：可视化(Visualization)、规格说明(Specification)、构造(Constructing)和文档化(Documenting)，下面分别进行说明。

1. 可视化

可视化能帮助开发者理解和解决问题，方便熟悉 UML 的开发者彼此交流和沟通。以此为基础，可以较容易地发现设计草图中可能的逻辑错误，保证软件能保质保量交付。

2. 规格说明

对一个系统的规格说明，应当通过一种通用的、精确的、没有歧义的通信机制进行。UML 适合这种说明工作。它使得编码前的一些重要的决定得以表示，使得开发者达成共识，能够对后续软件的开发过程进行指导，提高软件的开发质量，降低开发成本。

3. 构造

按照 UML 的语法规则，使用软件工具对模型进行解释和说明，将模型映射到某种计算机语言来实现，大大加快系统建模和实现速度。

在实现的过程中，根据设计，合理调配资源，并识别可复用的组件，高效实现复用，降低开发成本。

4. 文档化

使用 UML 可以同时生成系统设计文档。这些专业化的设计文档可以帮助开发者节省开发时间，快速熟悉和理解系统，起到人与系统之间的桥梁作用，达到事半功倍的效果。

UML 是提供给用户的高层建模的方法，是针对用户需求的高层抽象，具体表现为描述组件的构成及联系，给人一种高屋建瓴的效果。代码也能够描述一种模型，但是是具体的、底层的，如果其他人想要了解设计思想，必须要读懂代码，甚至可能要先去学习一门新的语言。还有一种表示方法是使用自然语言描述，但自然语言具有歧义和含糊不清的缺点。

UML 具有简单的表示法(Notation)，而且其定义是规范和严谨的。UML 的表示法简短，容易学习，而且其含义有明确的定义(在 UML 中，这种定义是通过元模型描述的)。

UML 2.0 的构成及其与 UML 1.x 的比较如下。

UML 1.x 明确了沟通设计、传达设计的要点，以捕获需求，并将需求映射到软件的解决方案。UML 2.0 经过体系的重建，克服了 UML 1.x 的过于复杂、脆弱和难以扩展等缺点，引进了一些新的图模型，以便扩展语言，使其能够支持最新的最佳实践。

UML 2.0 具体包括以下模型。

(1)用例图：用于表示系统与使用者(或其他外部系统)之间的交互，有助于将需求映射到系统。

(2)活动图：用于表示系统中顺序和并行的活动。

(3)类图：用于表示类、接口及其之间的关系。

(4)对象图：用于表示类图中定义的类的对象实例，其配置是对系统的模拟。

(5)顺序图：用于表示重要对象之间的互动顺序。

(6)通信图：用于表示对象交互的方法和需要支持交互的连接。

(7)时序图：用于表示重点对象之间的交互时间安排。

(8)交互概况图：用于将顺序图、通信图和时序图收集到一起，以捕捉系统中发生的重要交互情况。

(9)组成结构图：用于表示类或组件的内部，可以在特定的上下文中描述类间的关系。

(10)组件图：用于表示系统内的重要组件和彼此间交互所用的接口。

(11)包图：用于表示类与组件集合的分级组织。

(12)状态图：用于表示整个生命周期中对象的状态和可以改变状态的事件。

(13)部署图：用于表示系统最终怎样被部署到真实的世界中。

1.3 系统开发的解空间

在面向对象分析和设计方法中，因为 UML 是描述面向对象模型的标准化图形语言和表示法，本书将使用 UML 来完成面向对象分析和设计。

现代的面向对象分析和设计方法是基于模型的，综合使用用例建模、静态建模、动态建模和架构建模来描述软件需求、分析和设计模型，构成了系统开发的解空间。在用例建模中，系统的功能性需求按照用例和参与者进行定义；静态建模提供了系统的结构化视图，类按照其属性及其与其他类的关系进行定义；动态建模提供了系统的行为视图，对象交互用于表示对象之间的通信和协作，用来实现用例；架构建模是系统的核心，使用包、子系统和构件结构来描述系统的框架结构及框架中各个部分的连接关系。

本书描述的一种基于 UML 的软件建模和架构设计方法，如图 1.3 所示。用例作为整个分析设计的驱动，然后基于软件架构的理论和方法构造出软件的总体架构，以此为核心，进一步使用面向对象的静态和动态建模方法，完成以类为单元封装体和以构件为复合封装体的分析设计，最终生成代码，实现系统。整个过程是一种基于用例的高度迭代的软件开发过程。

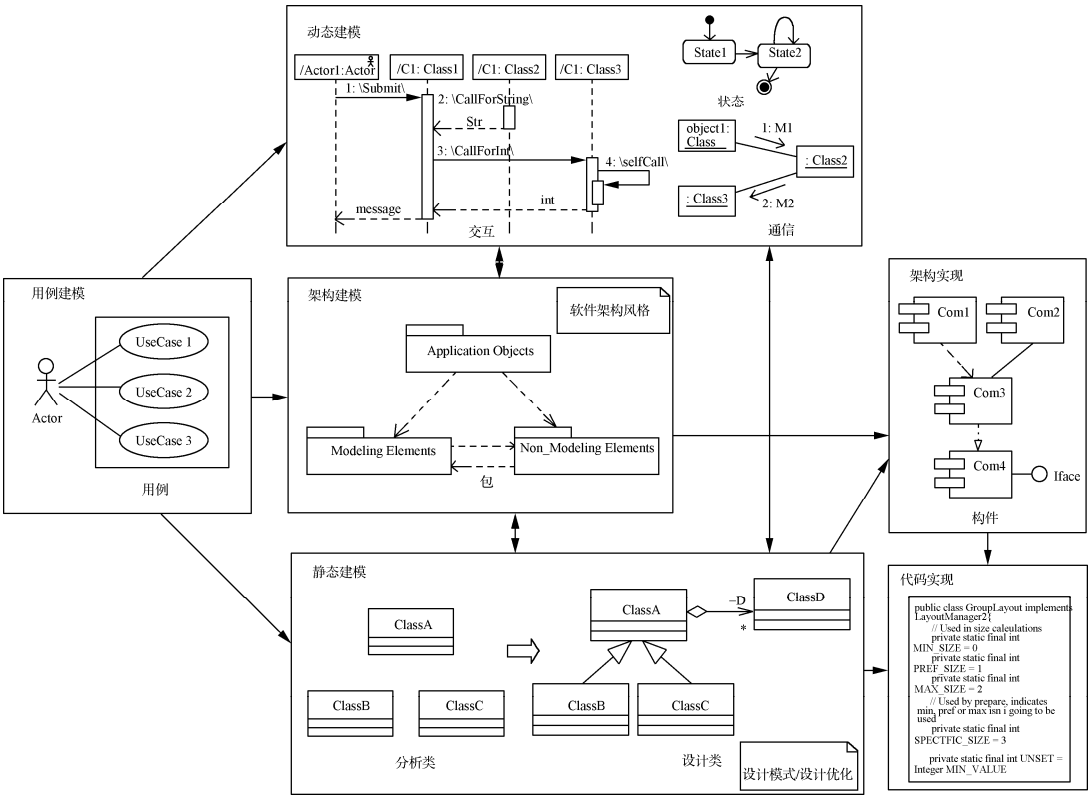


图 1.3　基于 UML 的软件建模和架构设计方法

1. 用例建模

在用例建模阶段，所开发的模型使用参与者和用例描述了系统的功能性需求，每个用例要开发一个叙述性描述的用例规约，在此过程中，用户的输入和主动参与是必不可少的。

2. 分析和设计

在分析和设计阶段，要迭代式地进行系统的静态建模和动态建模。静态建模定义了问题域中类之间的结构、关系，这些类及其关系描绘在类图中。然后，动态建模实现来自用例建模的用例，以显示每个用例中参与的对象及对象间是如何交互的，对象和它们之间的交互描绘在通信图或者顺序图中。在动态建模中，使用状态图来定义与状态相关的对象。

3. 架构建模

在架构建模阶段，要设计系统的软件体系结构。在此阶段中，模型被映射到一个运行环境中。问题域的分析模型被映射到设计模型的解空间中，此阶段提供包和子系统的组织准则来将系统组织为子系统。子系统被视为聚合或者复合对象，当分布式的子系统中使用相互通信的可配置构件进行设计时，要给予特别的考虑。

接着设计每个子系统。对于顺序系统，重点放在信息隐藏、类和继承的面向对象概念。对于并发系统，例如实时、客户端/服务器(C/S)或分布式应用，除了考虑面向对象概念，还需要考虑并发任务的概念。

4. 实现

在实现阶段，要将软件架构模型映射到可部署运行实现的模型解空间中。在此阶段中，包和子系统映射成组件，而其中的类则使用代码生成方法和优化策略生成可运行和部署的面向对象结构的代码。

1.4 软件工程开发方法

软件工程在软件开发过程中引入了一整套相关的技术及规范，主要包含方法、工具及过程三个基本要素。方法是完成软件开发各项任务的技术，主要回答“如何做”；工具是为方法的运用提供自动或半自动的软件支撑环境，主要回答“用什么做”；过程则是为了获得高质量的软件要完成的一系列任务的框架，规定完成各项任务的步骤，主要回答“如何控制、协调、保证质量”。随着软件工程的发展及技术的不断进步，根据人们分析问题角度和方式的不同，软件工程开发方法可以笼统地分为传统方法(结构化方法)和面向对象方法。

1.4.1 传统方法

传统方法又称结构化方法，是一种静态的思想，将软件开发过程划分成若干个阶段，并规定每个阶段必须完成的任务，各阶段之间具有某种顺序性。传统方法体现出对复杂问题“分而治之”的策略，但主要问题是缺少灵活性，规范中缺少应对各种未预料变化的能力，而这些变化却是在实际开发中无法避免的。因此，当软件规模比较大，尤其是开发的早期需求比较模糊或者经常变化时，这种方法往往会导致软件开发不成功。即使开发成功，维护起来通常也比较困难，增加系统的总成本。

1.4.2 面向对象方法

面向对象方法是一种动态的思想，其出发点和基本原则是尽可能模拟人类习惯的思维方式，将现实世界中的实体抽象为对象(Object)，对象中同时封装了实体的静态属性和动态方法。面向对象分析设计的方式使得业务领域中实体及实体之间的关系与对象及其关系保持一致，做到了概念层与逻辑层的相互协调，更要强调的是各种逻辑关系在结构上的稳定性[①]，通过稳定

① 这种稳定性可以通过 UML 的类图进行表达。

的结构来提高应对各种变化的能力。因此，它的开发过程可以是一个主动多次迭代的演化过程，保证了开发阶段间的平滑(无缝)过渡，降低了模型的复杂性，提高了可理解性及应对各种变化的能力，从而简化了软件的开发和维护工作。

技术上，对象融合了数据及在数据之上的操作，所有的对象按照类(Class)进行划分，类是对象的“抽象”；类与类之间可以构成“继承”的层次关系；对象之间的互相联系是通过消息机制实现的，确保了对信息的“封装”，使得对象之间更为独立。

同时，面向对象的分析过程既包含了由特殊到一般的归纳思维过程，也有由一般到特殊的演绎思维过程，而且对象是更为独立的实体，可以更好地进行“重用”。

1.4.3 理解两种开发方法

以上对传统方法和面向对象方法进行了介绍，下面通过一个具体的例子进行说明。后续章节还会对这个例子中涉及的具体方法和模型做更详细的叙述。

考虑日常生活中一个熟悉的应用场景“餐馆就餐”。传统的思维方式是一种过程化的方式，即将整个就餐过程划分成许多子过程，每个子过程对应不同的处理流程，数据在这些子过程中流动和处理，因此这种方法又称面向数据流的方法。图 1.4 形象地描述了运用两种不同的思维方式产生的不同分析方法。图中的下半部分描述的是传统的过程化方法的分析图[①]，整个就餐过程划分为点菜、备料、烧菜、上菜 4 个子过程：首先顾客借助菜单在点菜过程中表达就餐意愿并记录在点菜清单中，后厨根据点菜清单进行菜品的备料和烧菜，上菜后需要对点菜清单做适当的标记。通过这样的分析过程，整个业务流程被描述和理解，并会在进一步设计中按照系统工程的层次方法对这些子过程进行更加具体及技术化的展开。

面向对象的思维方式是以人类对现实世界的理解为出发点，对领域中的实体进行抽象，因此简化了分析的难度并增加了可理解性，同时保证了分析模型结构的稳定性，提高了应对变化的能力。图 1.4 中的上半部分是面向对象的分析方式，通过业务领域的理解抽象出 3 个主要的类，即顾客、服务员和厨师。图中标明了各个类具有的服务能力，如服务员主要是对顾客提供点菜和上菜的服务。值得一提的是，厨师和顾客类中都具有一个“品尝”的服务，但这个服务在两个对象中的作用是不同的。厨师的品尝服务前面有一个短横线标识，表明这是一个私有服务——仅在本对象内使用(烧菜需要的动作)。顾客类中的品尝则是一个公有服务。

抽象类之间的关系是保证整个分析结构稳定的重要元素之一，图中顾客与服务员、服务员与厨师之间的箭头表明了对象之间具有的联系，这种联系在这个特定的业务领域(餐馆就餐)中是很自然的，与我们日常的就餐场景一致，而顾客与厨师之间并没有任何联系，符合该业务的一般规则，并且是一种固定的业务模式，所以具有较强的稳定性及可理解性。

如果用户此时对需求提出了新的修改建议：需要当前这个就餐系统能够“礼貌待客”。需求变化的响应对于软件开发者来说一直是一个比较头疼的问题，应对需求变化的能力体现出的是设计师的素质和设计质量。针对用户具体的需求变化，传统的方法中必须首先要筛选出业务变更的位置——服务员与顾客之间的点菜和上菜这两个过程。不幸的是，上述隔离过程需要设计及维护人员熟悉和理解整个业务流程，对于一些较大型的系统，往往这是很困难的一件事情。

① 此图的名字叫做数据流图(Data Flow Diagram，DFD)。

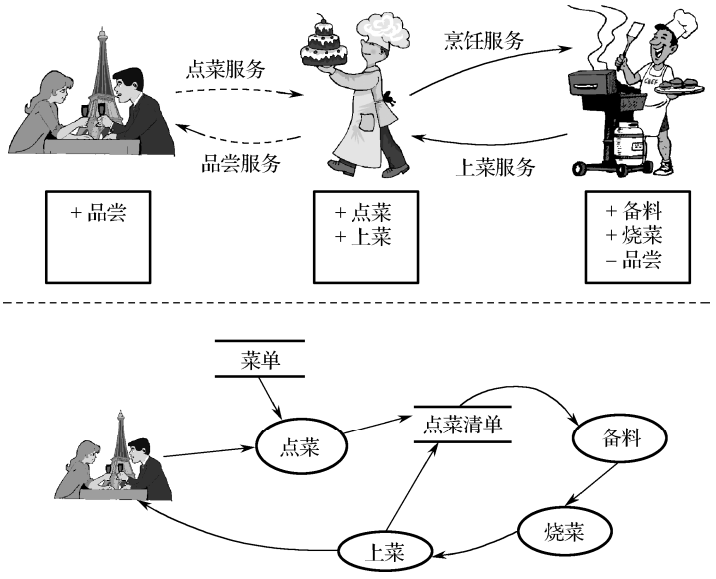


图 1.4　传统方法与面向对象方法的分析比较

面向对象过程在应对这一变化时是比较简单和直接的，由于其分析结果就是对现实世界的忠实反映，所以可以知道与礼貌待客直接相关的对象就是服务员，其他的对象根本不需要修改，因此也不需要去深入理解它们。由于类的封装性，我们可以直接把修改限定在一个特定的范围内，模块化的优势也就体现出来了。具体地，在本例中，可以在服务员类中添加一个私有的“问候”服务，并在该类内的点菜和上菜服务中使用此服务即可，其他地方及其他对象，尤其是对象之间的关系结构不用做任何变化，“隔离变化、应对变化、以不变应万变”正是面向对象方法优势的体现。

以上简单介绍了传统方法和面向对象方法的特点及其比较，面向对象方法无论是在理念上，还是在实际开发过程中，都比传统方法更具有优势，这也是面向对象方法在工业界越来越成为主流的原因。即使这样，也不能说可以完全摒弃传统方法。作为面向对象方法的补充，这些传统的、经典的分析和设计方法和过程目前仍然有着广泛的应用。

1.5　习题

1．软件工程主要包括哪些内容？

2．面向对象分析方法优于传统方法的根本原因是什么？可否借助图 1.4 或其他实例给出自己的理解？

3．UML 包含哪些重要的模型？它们在系统开发的解空间中作用如何？

第 2 章　软件开发过程

软件开发过程(Software Development Process)又称软件开发生命周期(Software Development Life Cycle，SDLC)，是软件产品开发的任务框架和规范，又可以简单地称为软件生命周期。软件生命周期有很多相应的模型，描述了开发过程中涉及的任务或活动的方法。本章首先介绍软件生命周期，然后针对软件生命周期中经常采用的传统生命周期模型和敏捷生命周期模型分别进行详细说明。

2.1　软件生命周期模型与软件过程

通常，软件生命周期模型更一般化(General)。例如，多种具体的软件过程都属于迭代式的软件生命周期模型。国际标准 ISO/IEC 12207 是软件生命周期模型的国际标准，旨在提供一套软件开发与维护过程中涉及的各种任务定义的标准，如软件生命周期的选择、实现与监控等。

可重复的、可预测的软件过程能够提升软件生产的效率和质量。过程改进是对过程本身进行不断的提高，引入有效的开发经验，是质量保证的重要环节之一。软件工程过程小组(Software Engineering Process Group，SEPG)在过程改进中具有重要的作用，提供给软件开发者统一的、标准的开发原则，确保每个成员都在做正确的事情，步调一致，充分协调各开发者、开发小组，通过过程控制的方法，保证软件产品的质量。

因此，软件生命周期模型是软件开发的宏观上的框架，软件过程则涉及软件开发的流程等管理细节，在框架稳定的前提下允许对软件过程进行裁剪。软件生命周期模型与软件过程的关系如图 2.1 所示。

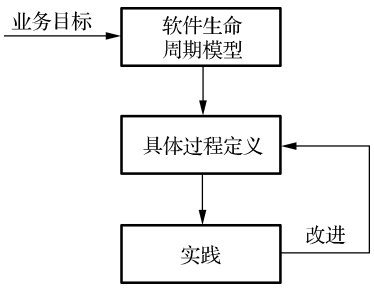


图 2.1　软件生命周期与开发过程

目前主要存在 4 种不同类型的软件生命周期模型，它们是顺序式、迭代式、增量式和敏捷式。在不同的软件生命周期模型中，其过程管理主要采用的是“分而治之”的思想，即将整个软件的生命周期划分成软件定义、软件开发和运行维护三个主要时期，每个时期再细分为具体的阶段，分别对应明确的任务，并在此基础上实现不同的管理方式。这样做的目的是使规模大、结构复杂、管理复杂的软件开发变得容易控制和管理。通常，软件生命周期可再细分为可行性分析与开发计划、需求分析、软件设计(概要设计和详细设计)、编码、软件测试、软件维护等阶段。

1. 可行性分析与开发计划

可行性分析的目的是用最小的代价在尽可能短的时间内确定该软件项目是否能够开发，是否值得开发，最后给决策者提供做与不做的依据。其实本质是要进行一次简化，压缩需求分析和设计过程，是在较高层次上以抽象的方式进行的一次需求分析和设计过程。

可行性分析的任务首先需要进行概要的分析研究，初步确定项目的规模和目标，确定项

目的约束和限制；然后进行简要的需求分析，抽象出该项目的逻辑结构，建立逻辑模型；最后从逻辑模型出发，经过压缩的设计，探索出若干种可供选择的主要解决办法，对每种解决方法都要从技术可行性、经济可行性和社会可行性三个方面进行研究。

技术可行性对要求的功能、性能及限制条件进行分析，以确定在现有的资源条件下，技术风险有多大，项目能否实现。资源包括软件、硬件、现有的技术水平、已有的工作基础。经济可行性即成本效益分析，估算将要开发的系统的开发成本，然后与可能取得的效益进行比较和权衡。社会可行性涉及的范围比较广，包括合同、责任、侵权、用户组织的管理模式及规范等。可行性分析结束时，需要提交一份可行性报告，报告中应描述所提出的解决方案和方案的可行性，并拟定一个粗略的开发进度。

开发计划是软件工程中的一种管理文档，主要对开发项目的费用、时间、进度、人员组织、硬件设备的配置、软件开发环境和运行环境的配置等进行说明和规划。开发计划是项目管理人员对项目进行管理的依据，据此对项目的费用、进度和资源进行控制和管理。开发计划文档一般包括项目概述、实施计划、人员组织及分工、交付期限等内容。

2. 需求分析

需求分析阶段是在确定软件开发可行的情况下，对目标软件未来需要完成的功能进行的详细分析。需求分析是软件开发后续阶段的基础，直接关系到整个系统开发的成功与否。由于用户的需求随着项目的进展不断变化，因此应采用合适的方法对需求变化进行管理，以保证整个项目的顺利进行，这个过程即需求变更管理。

此外，应充分理解和掌握用户对目标软件的期望，除功能性需求外，还要对系统设计有影响的非功能性需求加以识别和分析。需求分析阶段的输出是一份“需求规格(Specification)说明书”。

3. 软件设计

软件设计是在需求分析的基础上寻求系统求解的框架，如系统的架构设计、数据设计等。软件设计可分为概要设计和详细设计，此阶段的输出分别为“概要设计说明书”和“详细设计说明书”。设计方案是软件实现的蓝图，应综合考虑软件的性能、扩展、安全等因素，合理规划系统模块的结构，充分考虑未来变化的可能性，并预留空间，尽可能保证系统设计结构在整体上的稳定性。

4. 编码

编码是将软件设计的结果翻译成某种计算机语言可实现的程序代码。在编码中，必须制订统一的编码标准规范，尽量提高程序的可读性、易维护性，提高程序的运行效率。

5. 软件测试

程序编码后需要对代码进行严密的测试，以发现软件在整个设计过程中存在的问题，并加以纠正。软件测试可分为单元测试、集成测试及系统测试三个阶段。软件测试方法主要有黑盒方法和白盒方法。

软件测试过程需要建立详细的测试计划、编写测试用例、记录并分析测试结果，以保证测试过程实施的有效性，避免测试的随意性。

6. 软件维护

软件维护是软件生命周期模型中持续时间最长的阶段，是为软件能够持续适应用户的要求延续软件使用寿命的活动。软件维护包括改正性维护、适应性维护、完善性维护、预防性维护等类型。

下面，我们粗略地将软件生命周期模型划分为传统生命周期模型和敏捷生命周期模型两类。传统生命周期模型本质上严格按照开发顺序进行软件开发过程的组织，包括顺序式的瀑布模型和快速原型模型、增量式的增量模型、迭代式的螺旋模型和喷泉模型。敏捷生命周期模型注重更短的迭代和更少的增量，包括极限编程和 Scrum 等。

2.2 传统生命周期模型

一种最简单实用的软件开发过程的组织方式就是顺序地将生命周期阶段组织起来，这就是传统生命周期模型，其中具有代表性的模型有瀑布模型、快速原型模型。除此之外，本节还介绍增量式的增量模型及迭代式的螺旋模型和喷泉模型。虽然这些传统生命周期模型仍然存在很多问题，但它们目前仍然是基本的软件生命周期模型。

2.2.1 瀑布模型

瀑布(Waterfall)模型的构成如图 2.2 所示，该模型由于类似瀑布而得名。

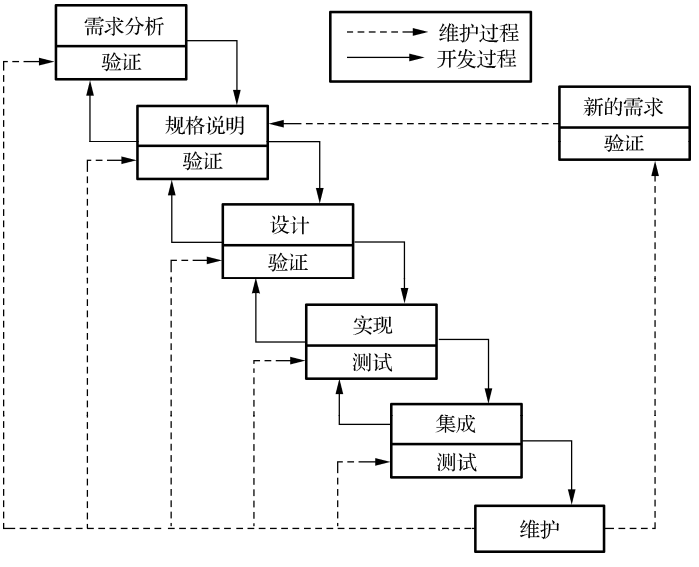


图 2.2 瀑布模型

1. 瀑布模型的特点

(1)文档驱动，只有当一个阶段的文档编制好，并通过评审之后，才可以进入下一个阶段。这是通过强制性的文档规格说明来保证每个阶段能够很好地完成任务。文档驱动是静态的开发形式。

(2)推迟实现，不急于编写程序，只有当对系统有了充分的认识和理解并完成了需求规格说明和设计规格说明后，才展开编码工作。

(3)质量保证，坚持各阶段结束前的评审活动，及早发现并解决出现的缺陷，避免延迟处理造成更高的代价。

2. 主要问题

(1)用户只有在开发早期及开发结束后，才有机会接触系统，这导致模型的能力天生具有缺陷，尤其是需求模糊或不明确的系统，开发过程中开发者有过多臆想掺杂，再经过各开发阶段的放大效应，使系统返工的风险大为增加。

(2)被动的救火式的应对问题，不希望有变化。变化来的越迟，付出的代价也越大，可谓“失之毫厘，谬以千里”。

(3)由于文档驱动式的开发方式，模型缺少灵活性，变更很不容易。而软件开发唯一不变的就是变化，瀑布模型(包括其一些变种)无法灵活应对变化的产生，导致其在应用上的局限性。

虽然瀑布模型存在很多问题，但其仍然是最基本的和有效的一种可供选择的软件生命周期模型。瀑布模型是一种计划驱动的模型，在对系统整体上的把控和协调上，具有一定的优势，因此，瀑布模型比较适合规模较大的系统开发或者分布式的开发模式。

2.2.2 快速原型模型

快速原型(Rapid Prototype)模型的结构类似于图 2.2，但需将图中的“需求分析”改为“原型开发”。快速原型主要作用是在用户和开发者之间，起到“桥梁”的作用。开发者和用户之间经常面临的一个状况是：用户熟悉的是业务但不懂得开发技术，而开发者正好相反，其更熟悉具体的开发方法、工具等技术内容，而不明白相关的业务流程，这是需求分析较难开展的原因之一，也是无法固化用户需求的客观因素。

1. 模型的特点

快速原型模型要求对系统进行简单、快速的分析，快速构造一个软件原型，用户和开发者在试用或演示原型过程中加强沟通和反馈，通过反复评价和改进原型，减少双方的误解，降低缺陷引入的概率，从而降低由于需求不明确带来的开发风险，提高软件质量，获取用户真正的需求。因此，快速原型模型比较适合一个全新的系统开发，用户借助原型能够了解到开发的方向是否正确，比较常见的做法是快速地构建用户界面，让界面首先体现出系统将来需要提供的功能布局，但界面元素下的真正实现可能并没有完成，因此主要起演示作用，但借助这样的原型系统，能够使用户建立起对未来系统的认识和了解，并和开发者逐渐达成共识。

另外，快速原型模型中可以尝试运用未来系统中需要的新技术，提前测试一些性能上的要求是否能够达到预期，如系统的运行或响应速度等，因此同样也降低了这方面的开发风险。

图 2.3 中给出的是一个实际项目的前期准备，目的是快速给用户提供一个“看得见、摸得着”的演示系统，寻求用户有价值的反馈和进一步开发的建议。这个过程强调的是功能的展示，而不是包含的内容或数据，例如，该原型系统中的数据并非实际要求的真实数据。



图 2.3　快速原型模型实例

2. 主要问题

快速原型模型的主要问题是所选用的开发技术和工具不一定是实际项目需要的。另外，快速建立起来的模型可能由于不符合各种开发规范，再加上不断的修改，质量一般都比较差，所以，通常在实际开发过程中会完全抛弃之前建立起来的原型系统。这也是快速原型模型被诟病的地方，如果采取某些措施和方法能够将快速原型在后续的开发中利用起来，对工期较紧张的项目还是非常有帮助的。

2.2.3 增量模型

增量(Incremental)模型又称演化模型，顾名思义，是指一步一步将软件建造起来。在增量模型中，软件被视为一系列的增量构件来设计、实现、集成和测试，每个构件是由多种相互作用的模块所形成的提供特定功能的代码片段组成的。

增量模型在各个阶段并不交付一个可运行的完整产品，而是交付满足客户需求的一个可运行产品的子集，如图 2.4 所示。整个产品被分解成若干个构件，开发者逐个构件地交付产品，这样做的好处是软件开发可以较好地适应变化，客户可以不断地看到所开发的软件，从而降低开发风险。但是，增量模型也存在以下不足：

(1)由于各个构件逐渐并入已有的软件体系结构中，因此加入构件必须不破坏已构造好的系统部分，这需要软件具备开放式的体系结构。

(2)在开发过程中，需求的变化是不可避免的。增量模型的灵活性可以使其适应这种变化的能力大大优于瀑布模型和快速原型模型，但也很容易退化为“边做边改”模型，从而使软件过程的控制失去整体性。

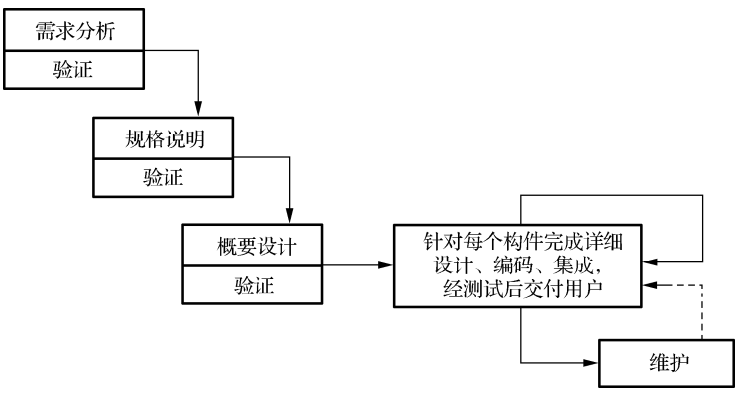


图 2.4 增量模型

使用增量模型时，第一个增量往往只实现基本需求的核心产品。核心产品交付用户使用后，经过评估，形成下一个增量的开发计划，它包括对核心产品的修改和一些新功能的发布。这个过程在每个增量发布后不断重复，直到产生最终的完善产品。

2.2.4 螺旋模型

1988 年，Barry Boehm 正式发表了软件系统开发的螺旋(Spiral)模型，它将瀑布模型和快

速原型模型结合起来，强调了其他模型忽视的风险分析，特别适合大型复杂的系统开发。螺旋模型沿着螺旋线进行若干次迭代，如图 2.5 所示，代表了以下活动。

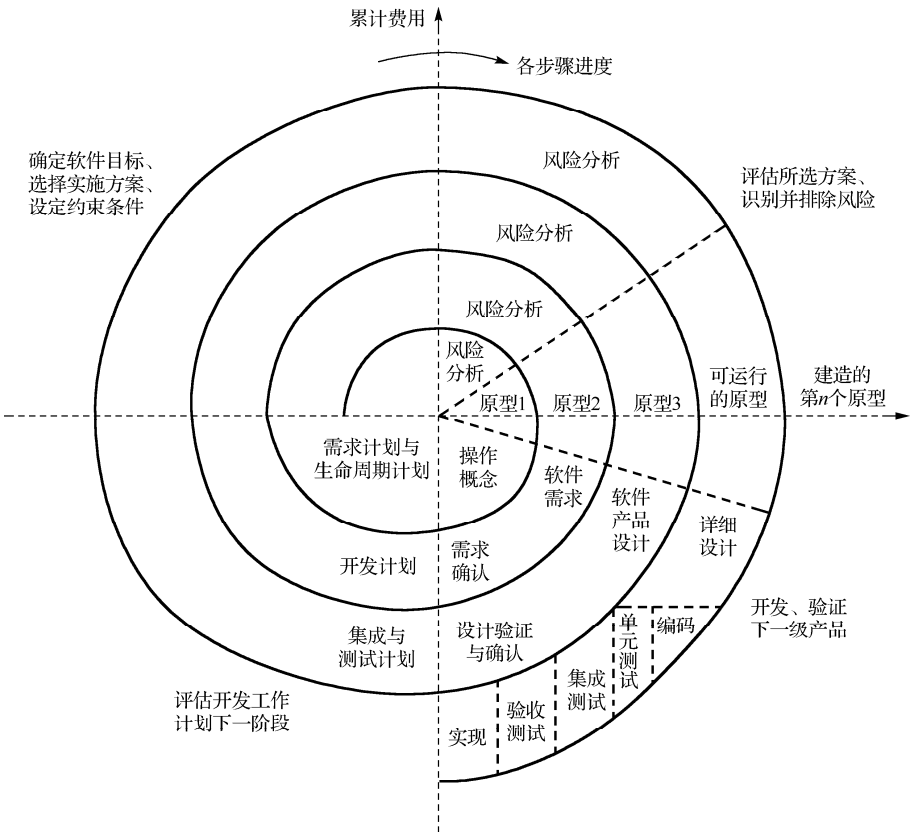


图 2.5　螺旋模型

(1) 制订计划：确定软件目标，选择实施方案，设定约束条件。

(2) 风险分析：评估所选方案，识别并排除风险。

(3) 实施工程：开发、验证下一级产品。

(4) 客户评估：评估开发工作，计划下一阶段。

螺旋模型由风险驱动，强调可选方案和约束条件，从而支持软件的重用，有助于将软件质量作为特殊目标融入产品开发中。但是，螺旋模型也有一定的限制条件，具体如下：

(1) 螺旋模型强调风险分析，但要求许多客户接受和相信这种分析，并做出相关反应是不容易的。因此，这种模型往往适用于内部的大规模软件开发。

(2) 如果执行风险分析将大大影响项目的利润，那么进行风险分析就毫无意义。因此，螺旋模型一般只适用于规模较大的软件项目。

(3) 软件开发者应该擅长寻找可能的风险，准确地分析风险，否则将会带来更大的风险。

该模型的每次螺旋迭代，首先应确定该阶段的目标，完成这些目标的选择方案及其约束条件，然后从风险角度分析方案的开发策略，努力排除各种潜在的风险，这个过程有时需要通过构造原型来完成。如果不能排除某些风险，那么该方案就立即终止，否则启动下一个开发步骤。最后，评估该阶段的结果，并设计下一个阶段的迭代。

2.2.5 喷泉模型

喷泉(Fountain)模型也称面向对象的生命周期模型。与传统的结构化生命周期模型相比，喷泉模型具有更多的增量和迭代性质，生存期的各个阶段可以相互重叠和多次反复，而且在项目的整个生存期中还可以嵌入子生存期，就像喷泉的水喷上去又可以落下来，可以落在中间，也可以落在最底部，如图 2.6 所示。

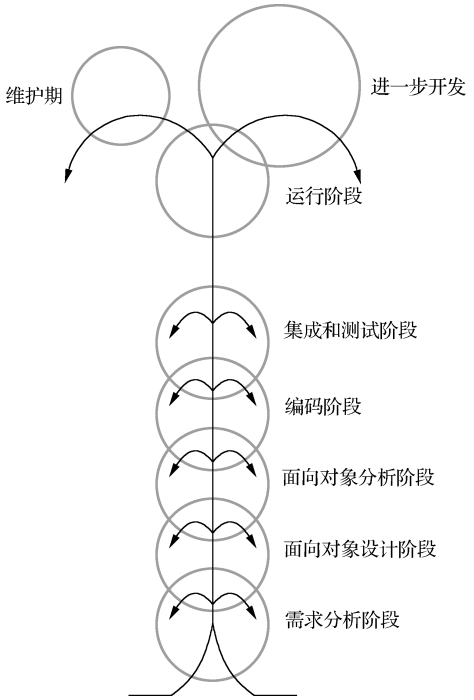


图 2.6 喷泉模型

面向对象可理解为直接面对问题域中客观存在的事物(领域概念、术语和关系等)来进行软件开发，它符合人们在日常生活中习惯的思维和表达方式。面向对象的分析过程直接针对问题域中存在的各项事物设立模型中的对象，问题域中有哪些值得考虑的事物，分析模型中就有哪些对象。因此，对问题域的观察、分析和认识是很直接的，对问题域的描述也是很直接的，所采用的概念和术语与问题域中的事物保持了最大的一致性。

同样，在面向对象设计阶段采用的也是与面向对象分析一致的表示方法，这使得从分析到设计不存在转换，只有局部的补充和优化，并增加与实现有关的独立部分，因此不存在传统方法中分析与设计之间的鸿沟，能够做到衔接紧密。

面向对象集成和测试同样也保持了这样的连续性。面向对象开发阶段间的无缝特性和表示方法的一致性，是喷泉模型优于传统软件工程方法的重要因素。

2.3 敏捷生命周期模型

敏捷开发是一种从 1990 年开始逐渐引起广泛关注的新型软件开发方法，是一种应对需求

快速变化的软件开发方法。很多敏捷生命周期模型的具体名称、理念、过程、术语都不尽相同，相对于“非敏捷”，更强调开发团队与业务专家之间的紧密协作、面对面沟通、频繁交付新的软件版本、紧凑而自我组织型的团队，能够很好地适应需求变化的代码编写和团队组织方法，也更注重软件开发中“人”的作用。

敏捷生命周期模型包括几种轻量级的软件开发模型，如极限编程(XP)、Scrum、精益开发(Lean Development)、动态系统开发方法(DSDM)、特征驱动开发(Feature Driver Development)、水晶开发(Crystal Clear)、开发运维过程(DevOps)等。不同的敏捷生命周期模型虽然在具体细节上不尽相同，但在方法论的层面上是一致的，它们遵循同样的原则：个体和互动胜过流程和工具；工作的软件胜过详尽的文档；客户合作胜过合同谈判；响应变化胜过遵循计划。

2.3.1 增量交付与迭代开发

增量交付与迭代开发是敏捷生命周期模型的两个基本特点。在增量模型中，我们介绍了增量开发的方式，即分批、分期地交付用户产品。无论是哪种类型的软件，只要需求确定了，就可以设计出理想的方案。但是，影响项目能否成功的因素太多了，如项目的工期、资金预算、人力资源等，开发者总是感到在种种因素的作用下，开发力不从心，其原因是完美方案的实现往往要付出巨大代价。软件开发具有很大的风险：市场情况、用户需求等外部因素都会发生改变，不及时发布软件和得到反馈，方案就无法得到验证，因此需要增量开发来应对软件产品之外的不确定因素。

与上次交付的产品相比，每次增量交付的产品都有新的功能。在规划版本时，一般常用的方法是按照功能的重要程度排序，但这样很容易陷入具体的细节中，使得系统的整体进度失去控制。敏捷生命周期模型给我们的建议是先实现必要的用户案例，体现出软件的价值，然后在后续版本中对功能进行细化，使得软件产品的所有功能都能达到相同的用户体验水平，如图 2.7 所示。用例(Use Case，又称用户案例或者用户故事)，是指用户通过系统完成的有价值的目标，它不是一个具体的功能。例如，用户按下计算器上的减号键，这是一项功能；用户按下“3–2=”的序列并计算出结果，这是一个用例。一个用例是用户与系统的完整交互，不同的用例可能会涉及相同的功能组合，但意义却不同。在用例中，如果用户按下的是“–3–2=”的按键组合，其中的减号表示的含义也是不同的。在敏捷生命周期模型中，用例通常会作为迭代的单位，这样每次交付的都是可以部署到用户应用环境中被用户使用的、能给用户带来即时效益和价值的产品。

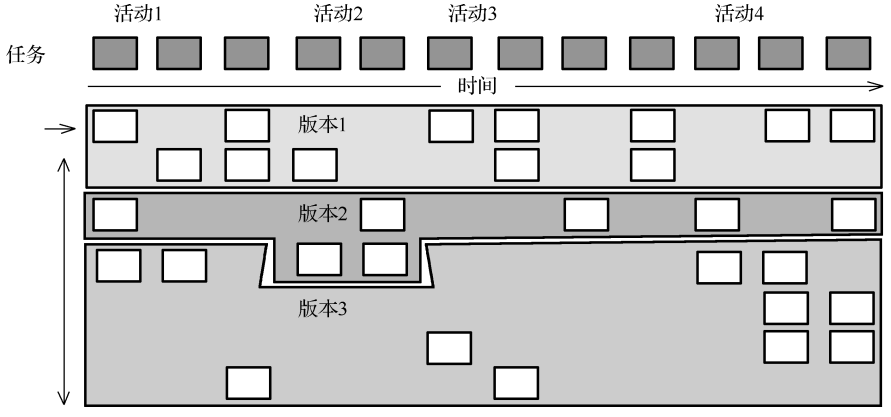


图 2.7　敏捷生命周期模型

在做需求调研时，经常会遇到一些客户，他们通常对需求陈述不清，或者只能大概交代目标系统的轮廓，无法描述细节，并且经常改变想法，需求无法稳定。迭代的思想就是，当我们对用户需求没有信心时，可以先构建后修改，通过多次反复，找到客户真正需要的软件。这是一个逐步求精的过程，一旦初始的方案是正确的，剩下的就是对方案不断提升和优化的过程，是个逐渐稳定的过程，如图 2.8 所示。

随着软件开发工作的深入进行，用户需求不断清晰，用例中的一些新的功能(必要功能、可选功能等)会不断地被发现。同时，用户也会不断成熟，提出更多新的需求。我们也可能受到市场的启发，得到了新的反馈。我们需要不断地丰富、细化用例，增加和改善新的功能。经过多次迭代完成所有的功能，从多个层次(必要性、灵活性、安全性、舒适性、趣味性等)满足用户需求，支持用例，从而逐渐缩小功能的不确定性，对功能的描述也越来越明确，这就是敏捷生命周期模型中的迭代过程，迭代效果如图 2.9 所示。敏捷生命周期模型中的迭代是客户和开发者共同成长的过程，每个迭代周期是一个定长或不定长的时间块，每个迭代周期持续的时间一般较短，通常为 1～6 周。

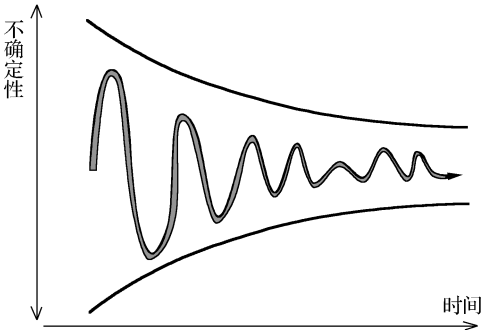


图 2.8　迭代稳定

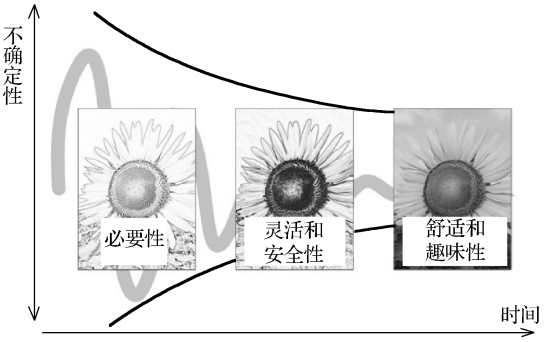


图 2.9　迭代效果

2.3.2　敏捷生命周期模型的优势

为什么瀑布模型多数情况下总会失败？为什么需要敏捷生命周期模型？这个问题在日新月异、飞速发展的今天似乎很容易解释。尽管瀑布模型能够在一个迭代周期内表现优异，但是在需求变化面前，瀑布模型却显得无能为力。事实上，大多数的软件项目都具有以下一些特点：

(1)在初始阶段，用户通常不能准确地知道他们需要什么样的软件。即便知道，也很少有人能准确、清楚地表达出来。

(2)对于某些项目，一开始，可以很好地定义其所有的功能，但是很多细节只能随着项目的不断深入，才能被挖掘出来。即便是我们了解了所有的细节，大多数人还是不能很好地处理这些细节，特别是在项目开发初期。

(3)外部环境，如客户的业务模式、技术进步，甚至是系统的终端用户，都可能在开发过程中不断改变，而预想或试图阻止这些改变通常都是徒劳的。

(4)在互联网时代，许多 Web 应用程序的开发都是基于对客户的远景预期，而非当前用户的实际需求。在这种情况下，变化从一开始就有，而且系统开始应用后几乎每天都会发生。

敏捷生命周期模型处理需求和技术变化主要通过增量和迭代过程。在每一次周期结束时，

都交付用户一个可用的、可部署的系统，通过让用户使用并体验该系统，开发者获得有价值的反馈意见，按顺序在随后的周期中和其他需求变化一起，在产品中实现和集成。每次迭代周期应尽可能短，以便能及时频繁地处理需求变化和用户反馈。

采用敏捷生命周期模型能够给企业和用户带来诸多好处。

(1)精确。产品由开发团队和用户反馈共同推动。瀑布模型通常会在产品起点与最终结果之间计划出一条直线，然后沿着直线不断往前走。然而，当项目到达终点时，用户通常会发现那已经不是他们想去的地方。而敏捷生命周期模型则采用小步的方式向前走，每一小步都是对真实目标的一次逼近，并及时调整下一步的方向，直到到达真正的终点。

(2)质量。敏捷生命周期模型对每一次迭代周期的质量都有严格要求。一些敏捷开发的过程中，使用测试驱动开发(Test Driven Development，TDD)，在正式开发功能代码前，先开发该功能的测试代码。这些都为敏捷生命周期模型提供了可靠的质量保证。

(3)速度。敏捷生命周期模型提倡避免较大的前期规划，认为那是一种很大的浪费。因为很多预先计划的东西都会发生改变，大规模的前期规划通常是徒劳的。敏捷开发团队只专注于开发项目中当前最需要、最具价值的部分，这样能很快地投入开发。

另外，新的功能或需求变化总是尽可能频繁地被整合到产品中。有些项目是在每个迭代周期结束时集成，有些项目则每天都在集成(Daily Build)。

(4)丰厚的投资回报率。在敏捷生命周期模型中，最具价值的功能总是被优先开发，这样能给客户带来最大的投资回报率。

(5)高效的自我管理团队。拥有一个积极的、自我管理的、具备自由交流风格的开发团队，是每个敏捷开发项目必不可少的条件。人是敏捷生命周期模型的核心，敏捷生命周期模型总是以人为中心建立开发的过程和机制，而不是把过程和机制强加给人。这既是采用敏捷生命周期模型的必然结果，又是推动敏捷生命周期模型不断前进的动力。

敏捷生命周期模型给企业带来了巨大的收益。据统计，采用敏捷生命周期模型的团队的开发效率一般会提高 3～10 倍，软件的质量也有了更加可靠的保证。同时，敏捷生命周期模型的应用也给团队内的每个成员提供了良好的发展机会，他们的技术和合作水平都能得到相应的提高。敏捷生命周期模型的成功来源于其方法本身的适用性和团队对它的深入理解和合理运用。当然，敏捷生命周期模型也不是万能的。通过以上分析可以看出，敏捷生命周期模型更适合规模中小、需求变化频繁的系统开发，强调团队的作用，所以更适合集中式的开发模式。

2.3.3 极限编程

极限编程(Extreme Programming，XP)的主要目的是降低需求变化的成本。它引入一系列优秀的软件开发方法作为开发实践的指导，并将它们发挥到极致。例如，为了能及时得到用户的反馈，XP 鼓励客户代表每天都与开发团队在一起。同时，XP 建议所有的编程都采用结对编程(Pair Programming)的方式。这种方式是传统的同行评审(Peer Review)的一种极端表现，或者可以说是它的替代方式。

XP 定义了一套简单的开发流程，包括编写用例、架构规范、实施规划、迭代计划、代码开发、单元测试、验收测试等。像所有其他敏捷开发方式一样，XP 预期并积极接受变化。XP 具有以下原则。

(1) 互动交流。团队成员不是通过文档来交流，文档不是必需的。团队成员之间通过日常沟通、简单设计、测试、系统隐喻及代码本身来沟通产品需求和系统设计。这里的隐喻与体系结构类似，都是从全局来描述一个项目的概貌，但隐喻不强调过多的符号和连接，而是定义一个从开发者到客户都能够理解的、全面一致的主题。隐喻可类比于拼图游戏中的原画，有了它的指引，开发过程也变得主动、有趣和可理解。

(2) 反馈。反馈是一种信息的交流，能使系统更加完善。反馈也和交流密切相关，客户的实际使用、功能测试、单元测试等，都能为开发团队提供反馈信息。同时，开发团队也可以通过估计和设计用户案例的方式将信息反馈给客户。

(3) 简单。XP 提倡简单的设计、简单的解决方案。XP 总是从一个简单的系统入手，并且只创建今天(而不是明天)需要的功能模块。因为它认为，创建明天需要的功能模块可能会由于需求的变化而成为浪费。

(4) 勇气。XP 鼓励一些应对较高风险的良好做法。例如，它要求开发者尽可能频繁地重构代码，删除过时的代码，不解决技术难题就不罢休。

(5) 团队。XP 提倡团队合作，相互尊重。XP 以建立并激励团队为一项重要任务。同时，它把互相尊重和实际的开发习惯相结合。例如，为了尊重其他团队成员的劳动成果，每个人不得将未通过单元测试的代码集成到系统中。因此，每个人的代码质量必须过关。

XP 注重可用于开发活动的实践，包括一系列核心的做法，也称最佳实践(Best Practices)。

(1) 小规模，频繁的版本发布，短迭代周期。

(2) 测试驱动开发(Test Driven Development)。

(3) 结对编程(Pair Programming)。

(4) 持续集成(Continuous Integration)。

(5) 每日站立会议(Daily Stand-up Meeting)。

(6) 共同拥有代码(Collative Code Ownership)。

(7) 系统隐喻(System Metaphor)。

2.3.4 Scrum

Scrum 是一个敏捷开发框架，与 XP 相比，其更注重软件开发的系统化过程。它由一个开发过程、几种角色及一套规范的实施方法组成。它可以被运用于软件开发、项目维护，也可以作为一种管理敏捷项目的框架。

在 Scrum 中，产品需求被定义为产品需求积压(Product Backlog)。产品需求积压可以是用例、独立的功能描述、技术要求等。所有的产品需求积压都是从一个简单的想法开始，并逐步被细化，直到可以被开发的程度。

Scrum 将开发过程分为多个冲刺(Sprint)周期，每个 Sprint 代表一个 1～4 周的开发周期，有固定的时间长度。首先，产品需求被分成不同的产品需求积压条目。然后，在 Sprint 计划会议(Sprint Planning Meeting)上，最重要或者是最具价值的产品需求积压被优先安排到下一个 Sprint 周期中。同时，在 Sprint 计划会议上，将会预先估计所有已经分配到 Sprint 周期中的产品需求积压的工作量，并对每个条目进行设计和任务分配。

在 Sprint 周期中，每天开发团队都会进行一次简短的 Scrum 会议(Daily Scrum Meeting)。

会议上，每个团队成员需要汇报各自的进展情况，同时提出目前遇到的各种障碍。每个 Sprint 周期结束后，都会有一个可以被使用的系统交付给客户，并进行 Sprint 评审会议(Sprint Review)。在评审会上，开发团队将会向客户或最终用户演示新的系统功能。同时，客户会提出意见及一些需求变化。这些将以新的产品需求积压的形式保留下来，并在随后的 Sprint 周期中得以实现。

随后，有一个回顾会议(Sprint Retrospective)，总结上次 Sprint 周期中有哪些不足需要改进，有哪些值得肯定的方面。最后，整个过程将从头开始，开始一个新的 Sprint 计划会议。Scrum 过程框架如图 2.10 所示。

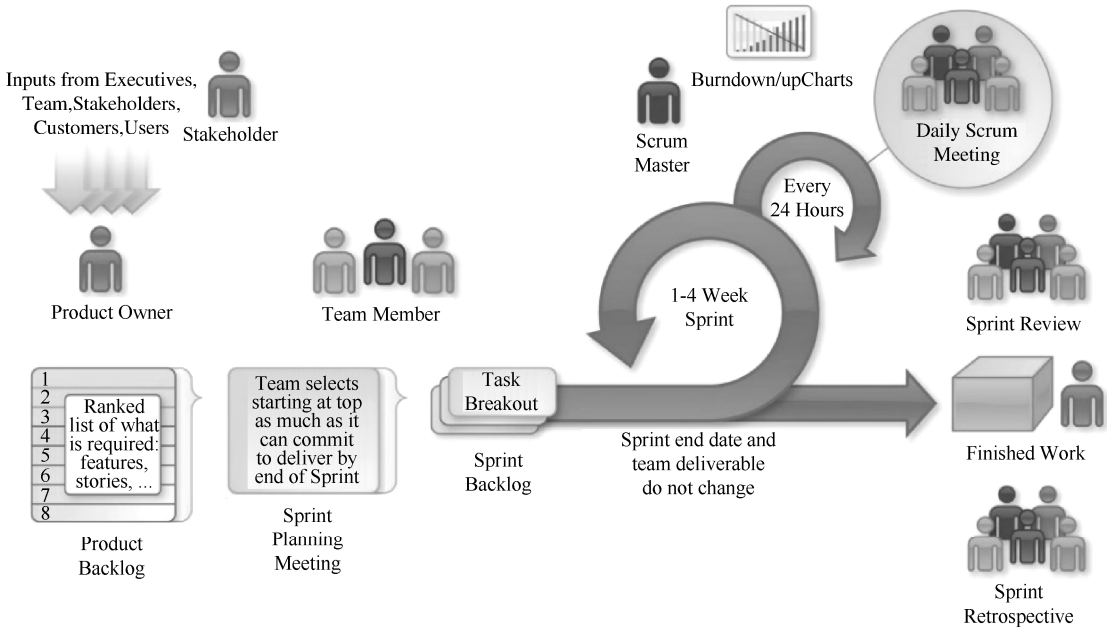


图 2.10　Scrum 过程框架

图 2.10 中的 Scrum 过程定义了 4 种主要的角色。

(1)产品拥有者(Product Owner)：负责产品的远景规划，平衡所有利益相关者(Stakeholder)的利益，确定不同的产品需求积压的优先级等。它是开发团队和客户或最终用户之间交流的联络点。

(2)涉众(Stakeholder)：该角色与产品之间有直接或间接的利益关系，通常是客户或最终用户代表。他们负责收集、编写产品需求，审查项目成果等。

(3)专家(Scrum Master)：专家负责指导开发团队进行 Scrum 开发与实践，是开发团队与产品拥有者之间交流的联络点。

(4)团队成员(Team Member)：即项目开发者。

Scrum 提供了一个敏捷开发框架，其他许多敏捷开发活动都可以被集成到 Scrum 中，如测试驱动开发和结对编程等。

2.3.5　DevOps 过程

DevOps 由开发(Development)和运维(Operations)两个单词组成，它源于敏捷开发，遵从

基本的敏捷原则，强调了“个体和交互胜过流程和工具”的作用，但不受限于某一种软件过程，甚至在瀑布模型中也会发挥作用。

DevOps 实际上是一组过程、方法与系统的统称，用于促进开发、技术运营和质量保证部门之间的沟通、协作与整合。它的出现是因为软件行业日益清晰地认识到，为了按时交付软件产品和服务，开发和运营工作必须紧密合作。例如，图 2.11 给出的是一个从开发到测试再到生产部署的一般流程，由于环境的不同，很难保证各个环节有一致的处理方式。如一个系统可能经常会有缺陷，但开发团队和测试团队会用不同的系统和方式来处理它们。这不仅在团队中导致了不必要的摩擦，并且把本应一起工作的双方隔离开了。运维团队很可能又会用不同的方式来处理服务器的部署请求。其实，三个系统都具有相似的工作流程，三个团队里的每个人都有使用一个相同系统的可能性，也许只需要为不同的角色展示不同的界面就可以了。如果三个系统变成了一个，就会带来减少维护成本的长期利益。

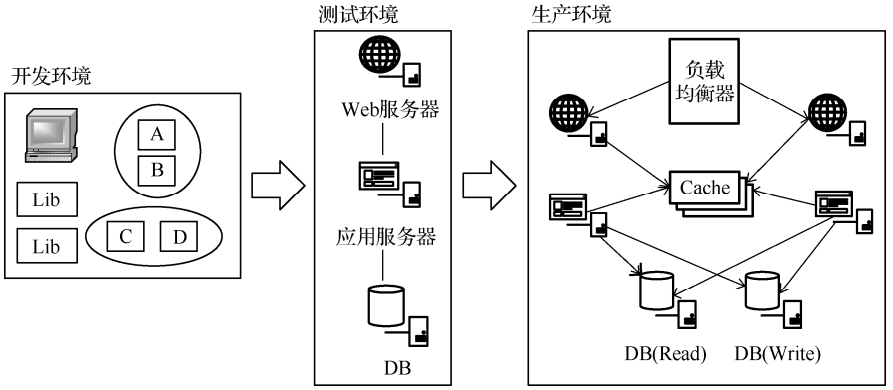


图 2.11　开发、测试与生产部署

DevOps 的另一个核心目标是自动化和可持续交付。简单来说，就是自动化一切可重复的、乏味的工作，把更多时间留给人与人之间的交流，产生真实的价值。DevOps 致力于让团队的流程更快、更有效、更可靠。只要有可能，就取代那些容易出错的、重复性的人力劳动。因此，DevOps 可以视为敏捷开发强有力的推手，它让敏捷开发的迭代周期更好地运转，实现敏捷企业中跨部门协作。无论是在短周期中还是长周期中，DevOps 同样会使它们变得更有效率。例如，DevOps 工程师维护的部署系统，让 Scrum 周期最后的交付更快、更有效。交付每 2～4 周就会周期性地发生，而在经常手动部署的企业里，部署一次可能需要好几天。在 DevOps 框架下，一旦代码提交到代码库，一个设计良好的、持续交付流水线大约只需要几分钟就可以把提交的代码部署到生产环境(取决于变更集的大小)。

综上所述，DevOps 提倡开发和运维之间的高度协同，打通开发和运维之间的部门墙，实现全生命周期的工具全链路打通与自动化和跨团队的线上协作能力，从而全面提高生产环境的可靠性、稳定性、弹性和安全性。DevOps 的过程模型如图 2.12 所示。

图 2.12 中的 DevOps 过程包含了两个维度的集成，即纵向集成和横向集成。纵向集成打通了应用全生命周期，强调的重点是跨工具链的“自动化”，最终实现全部人员的“自助化”。例如，项目组的开发者可以通过 DevOps 的平台自主申请开通需要的各种服务，如开通开发环境、代码库等。

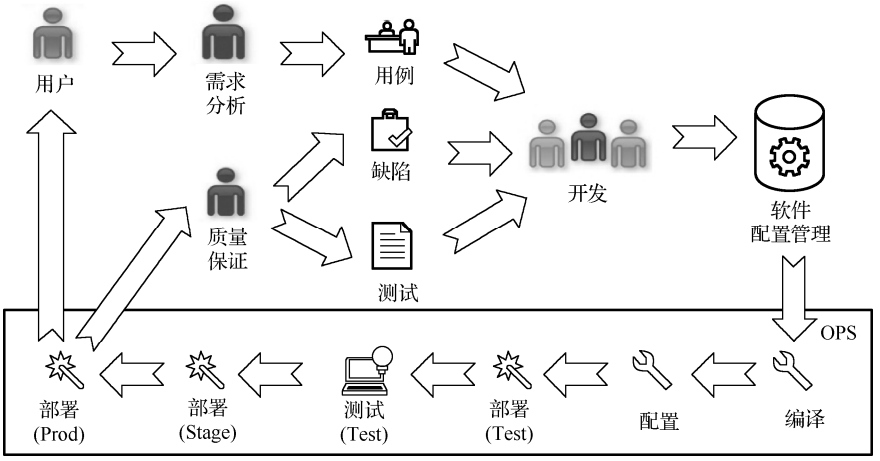


图 2.12　DevOps 过程模型

横向集成打通了架构、开发、管理、运维等部门墙。横向集成中 DevOps 强调的重点是跨团队的“线上协作”，即通过 IT 系统，实现信息的“精确传递”。例如，传统的系统上线部署方式，可能是一个冗长的说明文档，但在 DevOps 过程中提倡“一切皆代码”的理念，因此部署的方式应该是通过标准运行环境的选择、环境配置的设置、部署流程的编排，实现数字化的“部署手册”。这样的手册，不仅操作人员可以理解，机器也能够执行，过程可以被追踪和审计。DevOps 通过工具链与持续集成、交付、反馈与优化进行端到端整合，完成无缝地跨团队、跨系统协作。

2.4　习题

1．总结敏捷生命周期模型与传统瀑布模型主要的不同点及适用情况。
2．什么是用例？它与功能的含义有什么不同？举例说明。
3．简述 DevOps 的基本原理和任务。

第3章 需 求 分 析

需求分析是软件开发的第一个阶段。简单来说，需求分析的目标就是搞清楚用户真正想要的系统是什么，存在哪些约束条件。对于软件项目来说，这是一个至关重要的环节，关系着系统开发的成败。需求分析是整体上的输入，“垃圾入，垃圾出”说的就是这个道理，这也如同在一幢大楼的建设过程中，如果地基没有打牢，在建设的后期，即使采取了各种弥补的方法，也很难保证其质量。

本章主要围绕需求分析的主要方法和过程展开，对需求分析的捕获和描述、需求分析的细化、功能性需求和非功能性需求等方面进行了介绍。

3.1 需求分析活动

软件开发如同高楼建设，如果在开发早期引入了一个微小的问题，在后期也许需要几十倍的投入去修正它，到了这个地步，软件质量是无法保证的。毫无章法的软件开发，尤其是在开发过程中凌乱的需求分析，很容易导致整个项目失败。

需求分析是一个令人比较兴奋而又充满挑战的工作，因为开发者需要与未来系统相关的领域专家们在一起工作，去探索一个全新的未知领域。对于软件开发者来说，这是一个学习未知领域的过程，是一个值得享受和回味的过程。没有这个过程的铺垫，在后续的设计和实现等阶段中，就很难把握用户的真实预期，也更不会有灵感和创造力的产生。所以，需求分析是一个非常重要的阶段，为了顺利地开展这个阶段的活动，首先需要分析人员借助各种手段熟悉、了解和掌握相关的业务领域，这不同于简单的记录，对业务的理解，一定要经过充分的思考和分析整理，这样的解决方案日后才会奏效。

另外，每个人(包括领域专家们)都不会对系统的全局有一个完整的概念，每个人都是一个局部，很难预知哪些领域信息会对以后的开发产生影响；更进一步，开发者还要事事留心，客户的想法会在开发进行中毫无征兆地发生改变，如何采取措施来应对这些无法预知的变化，是需求分析的难点。

3.1.1 需求准备

需求分析阶段一般通过一份需求规格说明书的文档描述项目应该实现的内容。在需求分析的开始阶段，往往还伴随着一个较短的可行性分析活动。在可行性分析中，需要明确系统的功能性需求、边界位置，以及在技术、经济、法律或操作等方面，项目是否能顺利进行，是否存在潜在的风险，这些风险的影响程度和降低这些风险的措施，并给出一个合理可行的解决方案。可行性分析的结果还有另外一个作用，就是评估出系统初步的开发费用，这也是与客户(甲方)签订商务合同的依据。

初步的需求分析也可由项目的委托方组织相关的开发者实施，并给出系统主要的需求列

表。我们把以客户为主导制订的需求文档称为用户(业务)需求，而以开发者为主导制订的需求文档称为系统需求。两者有以下几个不同之处：

(1) 系统需求是对用户需求的细化和完善。

(2) 系统需求的阅读对象是开发者，而用户需求的阅读对象是委托方或客户。

(3) 系统需求是用户需求的开始，用户需求需要得到委托方的确认。

无论是系统需求还是用户需求，调研时都需要搞清楚项目涉及的主要目标人群及项目的主要目标。

3.1.2 系统涉众

涉众(Stakeholder)，又称利益相关者，是与目标系统相关的一切人和物，他们对目标系统的构建会有一定的影响。涉众是一个比较宽泛的概念，可以是提出系统原始需求的人，可以是委托方，也可以是使用目标系统的最终用户。有时涉众并不固定，很可能会突然出现在我们面前。例如，若某公司新增加了信息安全方面的要求，则信息安全负责人员会马上成为目标系统的涉众，他们会按照相关要求对目标系统进行检查，必要时会改变系统的设计策略和实现方式。

当然，对于影响系统的涉众来说，他们的重要程度是不同的。他们提出的需求有时可能是从不同的角度对目标系统的理解，甚至还可能有矛盾的地方。这就需要开发者不仅能够准确地将需求识别出来，并确定其优先级，而且能在他们之间斡旋和协调，当有矛盾发生时，说服他们，使其在某些方面妥协，让目标系统满足大多数涉众的要求，从而使系统开发得以向前推进。常见的涉众包括如下相关内容。

1. 最终用户

最终用户对系统成败具有举足轻重的作用，他们是系统的实际使用者，对目标系统有直接的接触和评价。为简化系统分析和设计，最终用户又会被划分为某些角色，不同的角色通常与系统有着不同的交互方式和交互内容。

这里的用户是一个抽象的概念，用户一般是涉众的代表，或者说，用户是目标系统的直接涉众。也就是说，用户是实际系统的参与者，是目标系统的一部分。其他涉众一般只在需求分析阶段用来分析系统，最终并不与系统发生交互。在系统的设计过程中，概念模型的建立及系统模型的分析一般都只从最终用户开始进行，不必理会其他涉众。

例如，在一个“软件项目管理系统”中，项目经理需要将任务指派给具体开发者；项目开发者要能够通过系统更新他们承担的任务信息；每个项目的进展情况和状态要能够及时通知项目经理和质量保证人员。以上这些人员和各自的角色都属于该系统的最终用户。

2. 投资者

投资者，又称出资方，主要关心的是目标系统的总成本、建设周期及未来的收益等高层目标。通常，投资者就是目标系统的业务提出者，但有时可能是不同的人或组织。例如，在一个“环保普查”项目中，省环保局可能是该系统的出资方，但并不负责该系统的管理和运营，具体需求的提出者和使用者是各地方环保单位。虽然出资方并没有直接关注系统的开发和技术，但不可否认的是，总成本、周期等因素会间接影响目标系统采用的技术及边界范围。

3. 业务提出者

业务提出者的目标通常是使现有的业务能够更加规范和高效，提升业务质量。例如，在“软件项目管理系统”中，投资者是某软件企业，其目的是提升公司内部开发过程的组织和管理水平、社会影响等，虽然这些期望都比较原则化和粗略化，但为该系统的开发者提供了建设宗旨和目标。在系统开发中，必须要将其贯彻，否则就会面临项目失败的风险。业务提出者通常是业务方的高层人员，如总经理，分管生产、财务、销售等部门经理等。

4. 业务管理者

业务管理者负责业务计划、生产、监督等环节的实际实施和控制。他们在业务方具有承上启下的作用，上对高层管理者负责，下对实际业务进行管理和安排。因此，他们关心如何方便地得知业务执行情况、如何下达指令、如何得到反馈、如何评估结果等。他们通常是业务方的中层管理者，如各科室的主任等。

业务管理者是需求分析过程中比较重要的调研对象，因为他们的需求相对更加具体，更为实际，他们对业务流程、业务规则及业务模式有着比较深入的理解。但也因为如此，调研过程需要进行科学的设计，引导他们将业务内容系统、浅显地阐述出来，帮助他们理清业务的主线和辅线，使得后续开发满足实际业务需求。

开发者和业务方总存在“沟通”上的障碍，开发者不懂业务，业务方不懂技术，这两个涉众在这个层面上的矛盾尤其突出和难以调和。因此，调研过程中，我们鼓励业务管理者和需求分析人员一同工作，业务管理者也可成为需求评审、系统部署等阶段的评价者和确认者。

5. 业务执行者

业务执行者是指实际的操作人员，是频繁与系统直接交互的人员。他们最关心的内容是系统会给他们带来什么样的便利，会怎样改变他们的工作模式。他们的需求最具体，系统的可用性、友好性、运行效率与他们关系最大。系统界面风格、操作方式、数据展现方式、录入方式和业务细节都需要从他们这里了解，他们也是系统能否成功的重心所在。

这类人员的期望灵活性最大，也最容易说服和妥协。同时，他们的期望又往往是不统一的，各种古怪的要求都有。他们的期望必须服从业务管理者的期望，因此，系统分析人员需要从他们的各种期望中找出普遍价值，解决大部分人的问题，必要时可以依靠业务管理者来影响和消除不合理的期望。

6. 第三方

第三方是指与业务关联的，但并非业务方的其他人或事。第三方的期望对系统来说不起决定性意义，但会起到限制作用，如标准、协议和接口等。

7. 开发方

开发方是合同乙方(受托方)的利益代表，他们关心的是这个项目是否有经济利益，是否能积累核心竞争力，是否能树立品牌，是否能开拓市场。这些期望将很大程度地影响一个项目的运作模式、技术选择、架构建立和范围确定。

8. 法律法规

相关的法律法规也是一个很重要的涉众，既指国家和地方法律法规，又指行业规范和标

准。法律法规通常会在业务领域的非功能层面上产生影响，对未来软件产品的适用范围或市场推广等方面起着较重要的作用，必要时应在开发合同中留下相应条款。

3.1.3 系统目标

当识别出所有的涉众后，接下来需要定义待开发系统的目标。在实际开发中，系统目标和涉众紧密相关，涉众提供了目标，同时目标也影响对涉众的取舍。

给出系统目标时要尽量将目标明确定义，并同时说明对这些目标验证的标准。通过对系统目标的整理，还要确保这些目标或目标的内部之间不会有矛盾的地方，整体上项目的各个部分都要保持一致。总之，系统目标的界定能够说明软件应该做什么，不应该做什么。

系统目标最直接的来源是系统应该提供的功能，这些功能通常不是单一的，对每个目标需要单独进行定义，可以借助表 3.1 所示的模板对各目标进行详细描述。

表 3.1 系统目标定义模板

项　目	内　容
目标	设置该目标可以满足的期望
对涉众的影响	对应的涉众及其影响
边界条件	附加条件或者约束
依赖	是否依赖其他目标，与其他目标的关系如何，是相辅相成还是此消彼长
其他	其他需要说明的内容

目标的定义是软件开发的纲领，是对用户需求进行深入分析的里程碑。用户需求只有与目标进行关联，才有实际意义。

3.2 用例与系统功能

下一步是确定与各涉众相关的系统核心功能。对于功能的识别过程，可以遵循完全不同的方法，这取决于项目的类型和涉众的计算机经验和技能。一些典型的方法如下。

(1)需求分析人员与相关涉众通过座谈的方式，共同确定未来软件支持的业务工作流程。需要注意的是，应顾及所有涉众的需求，尽量不要有遗漏。

(2)访谈关键的涉众人员，因为未来的系统最终由这些人员组织验收。此外，就预期的问题和风险，也要进行讨论。这种访谈形式非常适合规模较小的涉众团体或者难以联系的涉众代表。

(3)还可以采用调查问卷的形式，但是要求问卷中的题目一定要有代表性。同时需要注意，调查问卷不具有上述两种方法具有的交互性。

(4)采用上述方式与涉众讨论原系统或当前系统的运行情况及其适用性，重点讨论存在的各种问题，挖掘优化潜力。

(5)通过在现场对最终用户的调研和访谈，促进开发者身临其境的头脑风暴，使其逐渐融入开发过程，较为容易地理解业务过程的实现原理。

当然，上面给出的方法是可以结合使用的。需要注意的是，我们力求发现的功能是未来系统对外界提供的行为和响应，这些功能以后要转换成待开发的任务，但不涉及具体的实现步

骤。这些任务在项目进行中可以进行独立的分配，有明确的开始和结束时间(或者持续时间)，提供明确的输出结果，任务之间在逻辑上的依赖性也要分析并确定下来。

3.2.1 用例及其表示

长期以来，人们习惯使用一种交互的方式来描述系统的场景，借以“捕获”用户的需求，这就是用例的概念。这里对用户需求使用的是“捕获”一词，因为对于业务用例，不能只做静态的简单功能说明，而是要构建一个动态的场景，其强调的是参与者与系统的交互活动。在这样的交互场景中，相同的功能可能会有着不同的作用，如之前计算器例子中，减号键的功能在不同的场景中可能代表“减号”或者“负号”。通过用例的交互场景就可以更明确地捕获到用户的需求，再借助 UML 中的用例图，可以对这种活动进行文档化。Ivar Jacobson 在他的方法 OOSE 中倡导使用这种技术，并在面向对象开发领域中得到了广泛的应用。

用例是对需求深入分析和理解后的结果输出，用例的完善一般需要迭代进行，每次迭代都会添加对当前迭代中的一些业务细节。另外，每个用例具有文档化的说明——用例规约，其是对具体用例场景中业务流程的脚本式的说明。开发时，不求一次性完成某个用例的完全脚本化，先进行一些口头上的交流也不失为一种沟通的手段，然后逐渐细化，重要的是没有遗漏任何涉众关注的内容。

对于前面提到的“软件项目管理系统”，我们可以先使用图 3.1 所示的用例图进行简单描述，其中包含了 5 个主要的用例。每个用例使用椭圆来表示，代表一项用户与系统的交互。为了明确说明用例对应的涉众，用例通常直接或间接与某个人形符号(Stickman)相关联，称为 Actor。Actor 的英文解释为“演员”，是指使用系统的用户类别，不特指具体的人，因此按照角色(Role)来理解更合适。

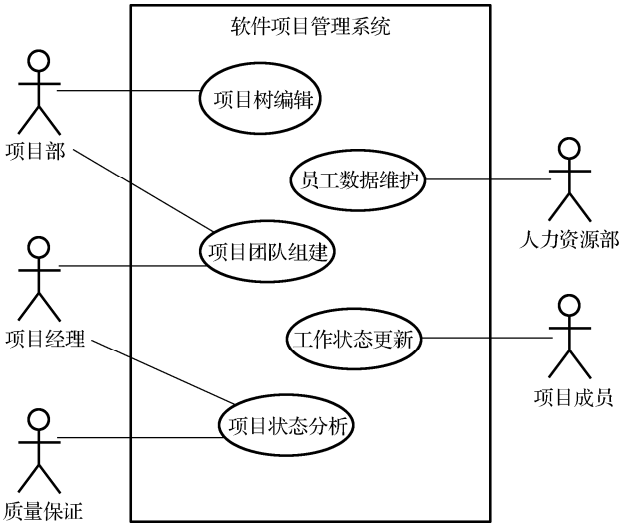


图 3.1 “软件项目管理系统”的用例图

不同的人可以扮演相同的角色，相同的人也可能在不同的场景中扮演不同的角色，角色与具体的人或职位并没有必然的对应。角色实现系统的某些用例，一个角色可以对应多个用例；相反，一个用例也可以对应多个角色。

3.2.2 识别用例

在实际应用中，我们可以先从角色出发来识别用例，角色一定是存在于系统边界之外的与系统发生某些交互的对象。掌握这条原则可以帮助我们更快速地确定用例图中的角色。

如前所述，角色不仅可由现实业务中的人来扮演，也可以是某些存在于系统之外的其他软件系统，如图 3.2 中“仓库管理系统”中的角色“零件目录系统”和“定时器(Timer)”。与“定时器”关联的用例“库存统计”表示该用例在特定的时间被自动触发，产生对该仓库的统计报告。

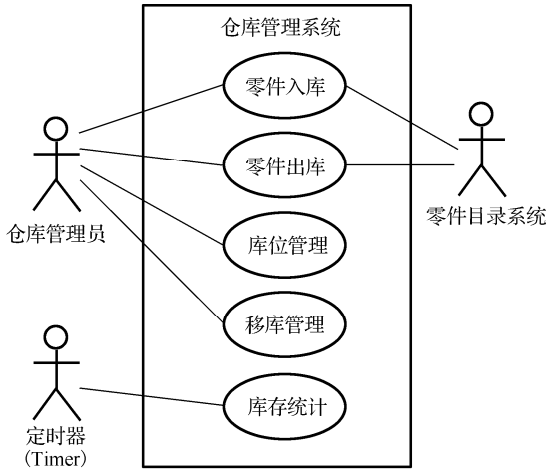


图 3.2　用例图中特殊类型的角色

寻找用例的过程实际上是一个具有创造性的过程，要围绕客户的预期，充分体现出用户需求。虽然这个过程非常灵活，难以掌控，但是可以考虑从以下几个方面进行切入。

第一，着重考虑在业务系统中进行管理和处理的一些关键业务实体，通常这是与业务相关的一些关键概念或技术术语。业务实体是业务管理中的基本信息，在高层的业务逻辑中与其他信息一起使用，构成更复杂的业务数据结构。例如，在“软件项目管理系统”中的业务实体主要有“软件项目”和“员工”两种。

(1) 软件项目(Project)：每个软件项目对应一个在系统中组织和管理的基本结构，项目开发者以团队的形式参与其中。

(2) 员工(Employee)：即开发者，是项目实施的基本单位，可进一步组成团队。

第二，对于业务实体，必须首先确定它们是由要创建的软件进行管理的，还是由已经存在的其他系统进行管理的。若它们由其他系统管理，则这些外部系统可识别为一种角色。如在项目管理系统中，若有一个外部系统负责管理员工的基本数据，则可加入一个“员工数据管理系统”的角色。

这里，假设对基本业务实体的管理都发生在系统边界内，则理论上可以直接识别出针对每个业务实体的 5 个用例，分别如下。

(1) 创建新的业务实体对象的用例。

(2) 修改已有业务实体对象的用例。

(3)删除已有业务实体对象的用例。

(4)持久化业务实体对象的用例，如存储到数据库中。

(5)访问业务实体对象的用例，如从存储的文件或数据库中读出数据。

如果这 5 个基本的操作业务实体对象的用例与同一个角色相对应，建议使用一个统一的用例进行表示，而不是使用 5 个单独的用例。例如，在上面的例子中，项目始终由项目部负责创建、编辑和删除，因此可用一个用例“项目信息编辑”来代表项目实体的增、删、改操作。同理，人力资源部门对员工信息的处理的绘制类似。另外两类用例涉及对数据的持久化操作，如果需要，通常使用单独的两个用例“备份”和“恢复”分别与之对应。当然，如果在数据创建和删除的同时会进行同步保存，后两种情况也就不用考虑了，因为这两个交互已经包含在增、删、改的用例中了。

第三，对同一事物尽量采用相同的专业术语进行指代，对用例命名时也要尽量具体和贴切一些，如“员工数据备份”要比“员工数据处理”更具体。用例的命名需要在与客户的沟通中逐渐清晰化，同时，用例是从使用者的角度来观察系统，与用户之间保持紧密的关系至关重要，也要尊重用户的习惯和选择。

第四，用例寻找需要围绕涉及业务过程数据的动态信息，它们是基本业务实体的组合和加工。业务过程数据是比业务实体数据更复杂的数据，而且它们会在业务执行期间频繁和动态变化以支持业务。例如，软件项目和员工两个业务实体通常只需要创建一次，在基本的维护过程中可能会有部分的修改，但相比之下的开发团队信息则改变得更频繁，这样才会适应业务中新的工作安排和调动。由此，项目团队和项目安排两种业务过程数据被识别出来。

(1)项目团队：对人员进行组织的单位，会被分配到具体的项目中。项目团队的构成情况在项目进行过程中会经常发生改变。

(2)项目安排：项目的起始和结束时间等，开发者需要按照调度安排开展工作。

被识别出的这些动态的业务数据结构同样可以按照上面的提到的 5 种情况进行分析并建立对应的用例，如“项目团队组建”和“项目安排更新”两个用例。在这一步发现的用例通常是业务的核心用例，用来描述系统的主要功能性需求，一般在访谈过程中由用户直接提出。

第五，需要考虑系统的功能性需求，它们也是系统的真正价值所在，一般是在识别出的基本业务实体数据和动态过程数据的基础上做进一步的计算和利用。如果一个系统只需要完成上述数据的维护工作，当然就不用额外添加其他用例了，但一个有价值的系统需要利用这些数据开展更复杂的业务活动，如对各种数据的组合处理、分析挖掘等。每个这样的数据利用，都会导致新用例的产生。在项目管理的例子中，“项目状态分析”用例需要对进行中的项目数据做进度上的分析和评估，如果形成了项目延期的结论，管理上就需要施加措施，如增加额外的开发者，而这可以通过已识别的用例“项目团队维护”来实现。对于更复杂的系统，系统中用例的数量会随着业务要求增加，如项目后期的计划变更等。

最后，需要考虑是否存在与正在运行的其他系统的交互。在分布式的环境中，与其他系统的交互，如同步或者监听数据等场景，都要设置单独的用例来捕获其功能性需求。

经过以上分析，可以将发现的用例记录下来，形成初始版本的用例图。用例图需要进一步补充和细化，可能的话，尽可能与客户一起工作，防止遗漏和误解，应忠实体现他们的需求。同样，客户也会理解开发者的开发过程，从而更容易接受开发者的工作成果。

从客户的角度形成的用例通常称为业务用例，是一次涉众与业务目标之间的交互，为涉

众创造价值。业务描述的是业务过程，而不是软件系统的过程，所以可以超越未来系统的边界。作为开发者来说，需要以业务用例为基础，整理归纳系统用例。系统用例是从计算机系统的角度描述业务系统，其业务边界就是系统本身的设计范围，角色是系统的参与者，与计算机系统共同实现一个目标。

这两种用例类型其实是密切相关的，一般来说，业务用例和系统用例存在一定的对应关系，系统用例描述了对应业务用例在软件系统中的实现。此外，往往还需要添加一些额外的系统用例，用于数据的初始化和维护工作。

例如，本例中的“项目团队组建”用例，这个业务用例用来描述在制订项目计划时相关部门的领导对计划的安排进行商讨，确定哪些开发者以何种角色参与到具体的项目中。在对应的系统用例中，则描述为项目部的负责人员利用系统浏览员工的工作状态并挑选合适的人选，然后为挑选出来的人员分配角色，并将其分配到开发项目中。

业务用例的建模一般适用业务活动比较复杂、涉及人员众多并需要长期深入某个行业的情形，而对于专业性和技术性较强的工具软件及应用软件等，大多数不需要业务建模的过程。本书主要关注系统用例模型的建立，因为对于“软件项目管理系统”这个例子，其需求的描述对于业务的理解已经比较充分了，所以，本书采取的方法是针对业务说明，在用例图中直接创建其对应的系统用例。

3.2.3 用例规约

用例规约是对每个用例的细化，用例规约的描述可以参照表 3.2 所示的模板。中间一列的数字为迭代的编号，表示在哪一轮迭代中应该完善此信息。在初始迭代中，只需为用例提供迭代编号为 1 的信息，然后在后续迭代中逐渐对上次迭代的信息进行补充，并完善当前迭代中的内容。

表 3.2 用例规约模板

项　目	编　号	内　容
用例名称	1	简短、精炼的描述，一般为动宾短语的形式
用例编号	1	项目中唯一确定的数字编号
包	2	在较复杂的系统中，用例被划归为不同的业务子系统，可以使用 UML 的包进行封装
维护者	1	创建和维护该用例的人员
版本	1	当前用例的最新版本号，或者将版本变化的历史一同保留，记录修改时间、目的和修改人
简介	1	简短描述该用例通过何种方式实现了什么功能
参与的角色	1	参与该用例的角色(涉众)，该用例对应的原始期望者
业务支持者	1	负责解答该用例对应哪些业务人员，分别处在哪些领域。如果需要修订，确认谁可以决定该用例的业务内容
引用	2	指出对该用例的实现有影响或有联系的信息来源，可能是某些规则、标准或现有的文档
前置条件	2	执行用例前系统必须所处的状态或达到的条件要求
后置条件	2	用例执行完毕后，系统可能处于的状态或结果
基本事件流	2	该用例正常执行的步骤和流程
备选事件流	3	基本事件流中异常或特殊情况的处理流程
关键性	3	该用例在系统中的重要程度
关联用例	3	其他相关的用例
功能性需求	2	有哪些具体的功能性需求是从这个用例派生的
非功能性需求	2	有哪些具体的非功能性需求是从这个用例派生的

对于用例中基本事件流和备选事件流内容的确定，以及派生的功能性需求和非功能性需求的归纳，后续的章节中还会进一步阐述。另外，在对用例进行文档化规约的同时，还应该建立一个词汇表，又称数据字典，用来描述相关业务术语的定义，要尽可能对相同的业务概念使用相同的术语进行描述，这对于需求的理解很有意义。

本例中的用例“项目信息编辑”，经过三次迭代和优化后的用例规约按照表 3.2 模板的格式可描述为表 3.3。

表 3.3 “项目信息编辑”用例规约

用例名称	项目信息编辑
用例编号	U01
包	—
维护者	杨楠 需求分析师
版本	1.0，2018 年 12 月 2 日，建立
简介	项目部人员有权编辑项目信息及其层次结构，包括项目各种属性、包含的子项目等
参与的角色	项目部人员
业务支持	王云 项目部主任
引用	《项目开发规范》
前置条件	软件成功安装并启动
后置条件	对于项目及子项目的修改成功提交系统
基本事件流	1．用户选择项目信息编辑功能 2．用户输入项目的基本信息 3．用户新建子项目 4．用户提交数据
备选事件流	用户可以选择已有的项目进行编辑 用户可以修改子项目的信息
关键性	高等级，系统核心功能

3.2.4 用例提炼

在用例图中，有时需要定义一些通用的过程，这些过程为其他主用例提供某种共同的基础性的功能，为避免这些相同的基础性功能被多次重复实现，可以将它们作为一种附加用例表示，并通过构造型<<Include>>与依赖它们的主用例相连。

在上面的例子中，系统要求每个主用例在执行之前必须成功验证每个用户的合法身份，这个登录过程对应一个通用的用例，可以将其提取出来，并通过图 3.3 的形式描述。

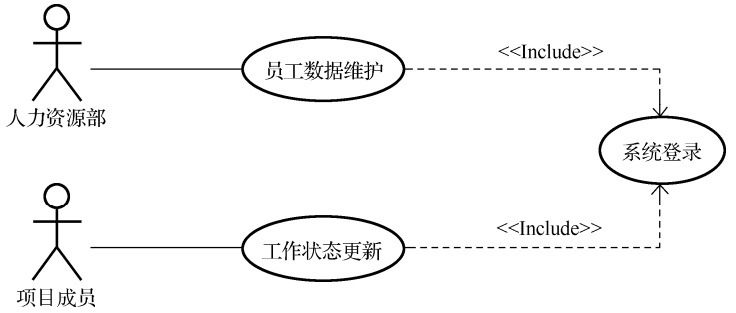


图 3.3 用例图中的包含关系

图 3.3 中新添加的用例“系统登录”为两个主用例提供基础功能，其表现形式与普通用例的椭圆形一样，也可以在这个用例上使用构造型<<Secondary>>，表示此用例是在细化的分析阶段被识别出来的(主用例的内嵌用例)。在 UML 的元素中，双尖括号用来表达此元素额外的属性或附加的语义，称为构造型(Stereotype)。在 UML 中，预定义了很多构造型，除此之外，用户也可以根据需要自定义与业务相关的构造型，从而对 UML 进行扩展。

包含关系<<Include>>需要通过文字的方式记录在用例规约的“关联用例”字段中，在基本事件流的对应描述中也应指明主用例使用其他用例的位置和方式。

<<Include>>的作用是避免某些交互过程的多次重复建模，但不应过分使用<<Include>>，尤其是不应该将用例错误地按照功能进行分解，把它们按照层次结构简单地组织，其中的某些用例只是作为连接其他用例而存在，其实质不对应具体的任务。图 3.4 描述方式就是一种错误的用例关系表示，按照功能分解的思想把“员工数据维护”划分为了“新建员工记录”“修改员工记录”“删除员工记录”三个子功能。需要说明的是，这里并不是由于引入了三个子功能而产生问题，其根源在于它们与主用例其实不处于相同的业务级别，业务粒度的程度不同。

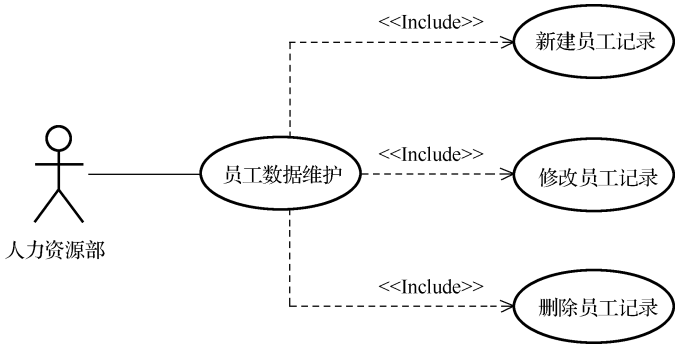


图 3.4　错误的用例关系

因此，在与客户一起进行业务建模时，在开始阶段应考虑完全不使用<<Include>>关系组织用例，因为这样可以集中精力，快速在模型中捕获参与者关注的业务场景，未来再考虑通过<<Include>>关系对这些用例进行细化和再加工，基本的原则就是保证图中的各个用例处于相同的业务级别。

在用例图中，两个用例之间还存在一种扩展的关系，描述了用例之间另外一种特殊的语义，在 UML 中以构造型<<Extend>>进行区分。

图 3.5 中描述的就是两个用例之间的扩展关系，强调了客户期望在某些情况下要表达出的一种意愿，此处表示在项目状态分析过程中，如果识别出一些异常的状况，并且增加了项目延期的风险时，应该由系统施加相应的措施，如警告提醒。图 3.5 中还存在另外一种图元，它可以出现在任何 UML 图中，即在一个右上角带有卷边的矩形中写有的注释文本。注释的文本有两种形式：非正式的文本形式和正式的形式化语言，图中是非正式的文本形式，将某种条件使用了花括号括起来表示，这也是 UML 中常用的表示形式。

扩展关系与包含关系很容易混淆，需要设计人员仔细甄别正确的业务含义及其联系。一般来说，被包含用例属于无条件发生的用例，而扩展用例属于有条件发生的用例；被包含用例提供的是间接服务，扩展用例提供的是直接服务；而且扩展用例在用例规约中一般作为备选事件流存在。

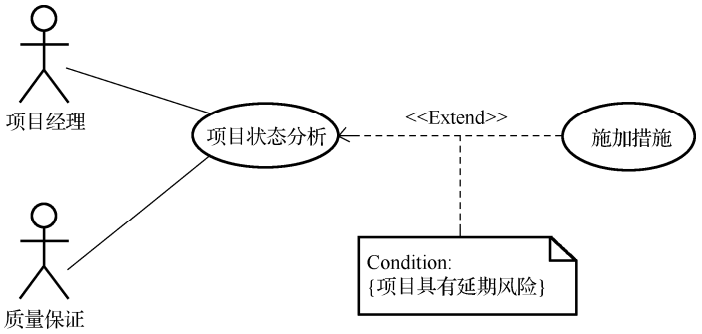


图 3.5　用例图中的扩展关系

对于<<Include>>和<<Extend>>两种构造型所代表的不同用例关系和使用场合，这里给出一些有意义的建议和方法，来尽量保证用例分析的正确性。一个好的用例能够提炼出明确的业务过程，能够给出清晰的业务起点和结果的定义。即使是在非常复杂的系统中，每个用例图中的用例数目一般要避免超过 15 个。同一用例图中的用例应尽量具有相同的业务粒度水平，并处于同一抽象级别。由于在实际设计中，上述的条件在形式上难以掌控，建议最终将所有用例放在一起进行筛选和分级。

一旦所有的用例具有了一致的形式，并使用了相同的业务词汇，这些用例可以为后续的开发提供基础，如进行进一步的开发和细化；以此为根据，进行较详细的成本估算；作为增量或迭代计划的基础单元；进行工作任务的分工等。

3.3　过程建模与事件流

前面给出的用例规约模板中，有两个使用文本方式描述的事件流：基本事件流和备选事件流。对于处理流程的建模，使用 UML 中的活动图是非常合适的。本节将介绍使用活动图进行过程建模的方法及应用。

3.3.1　过程建模

活动图的一般描述形式如图 3.6 所示，过程的每个步骤在图中都用带有圆角的矩形表示，称为动作(Action)，动作的执行顺序由动作间的箭头方向进行指示。图中黑色实心圆圈表示活动图的开始，带有外环的黑色实心圆圈表示活动图的结束。图中的菱形表示控制点，有两种形式：一种是多个箭头从控制点引出，表示分支的选择，在每个引出的分支上给出对应的条件，同时要求最多只能有一项满足，而且最好不要出现所有条件都不满足的情况，因为这样过程会在此处暂停并处于等待状态，直到某一个条件满足才会继续。另外一种是多个分支的汇入，表示不同的条件分支的汇聚节点。

图 3.7 是在活动图中对并行处理的一种表达方式，图中带有唯一一个输入箭头的黑色长条表示一个分支(fork)，它代表从此处开始可以有多个并行执行的子流程，例子中的两个并行流程分别是“选定项目经理”和“选定项目成员”，它们在执行的顺序上不分先后。图中另外一种只有一个输出箭头的黑色长条表示各流程的汇聚节点，汇聚的类型可通过一个花括号具体说明，例如，图中的{and}表示必须等待所有分支结束后才可以继续进行，即具有“同步”各分

支流程的作用。若是{or}，则表示有一个流程分支结束即可，这时其他分支中的活动会继续进行，而汇聚后的过程也会继续，并且不受那些并行流程结束的影响。

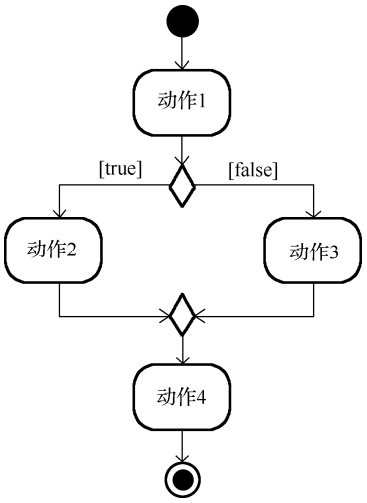


图 3.6　活动图

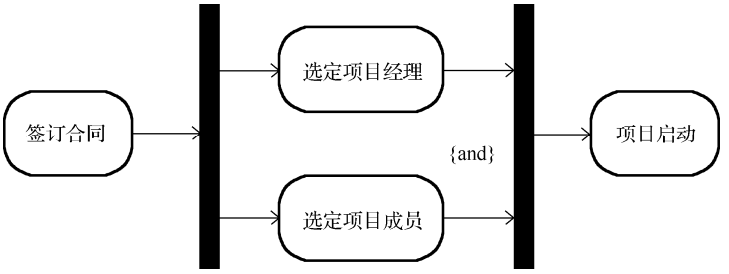


图 3.7　活动图的并行流程

在活动图中，除了动作，还可以嵌入表示数据的对象，这样就可以有对象流了，使得活动图的描述能力更强。对象流一般用来强调上一个动作的输出结果或者下一个动作需要的输入。图 3.8 所示的活动图描述了一个 Eclipse 项目的分析模型的创建过程，图中用矩形框标识了参与其中的对象，其状态对应方括号中的条件，例如，输出对象“Eclipse 项目[已创建]”会作为后续动作的输入继续处理。

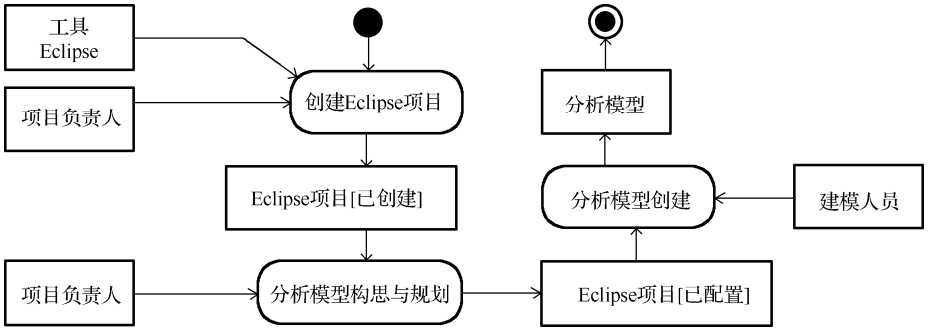


图 3.8　带有对象的活动图

活动图中的对象是可选的图元，在一些上下文较清晰的业务中，可以将这些对象类型在图中省略，以简化模型。过程模型的构建需要在充分了解需求的基础上，才能准确地识别出业务动作及其逻辑关系，也为日后软件的开发提供了依据和方向。

过程模型一般都采用增量的方式进行构建，首先描述出典型的流程，如果所有人员对过程的描述达成了一致，那么再逐渐加入其他可能的分支流程。每个步骤也可以是增量的，逐步加入可选的动作。当所有内容都充实后，可考虑对某些部分进一步提炼为单一动作，从而对模型进行简化和优化。下面通过一个典型的销售业务流程来具体说明其活动图的构建过程，我们模拟某公司的销售业务流程，并且按照增量的方法逐渐细化。

首先对典型的销售业务流程进行建模，如图 3.9 所示。对应的文字描述为：销售代表首先与客户取得联系并对其进行访谈，确定客户的具体需求，相关部门提供预算，然后与客户签订合同。从以上流程的分析中可以识别出的角色包括销售代表、客户及可能的相关部门(如技术部门)，涉及的主要对象包括客户的具体需求、预算及合同。

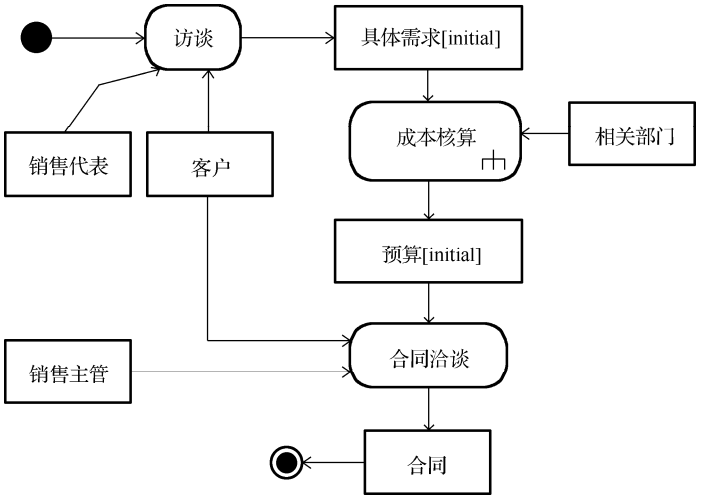


图 3.9　典型的销售业务流程

在接下来的迭代中，我们要对销售业务流程进行深入的研究，加入业务的细节内容。作为示例，假设客户有可能对预算的成本不能接受，然后调整需求，并重新核算成本。这个过程如图 3.10 所示。

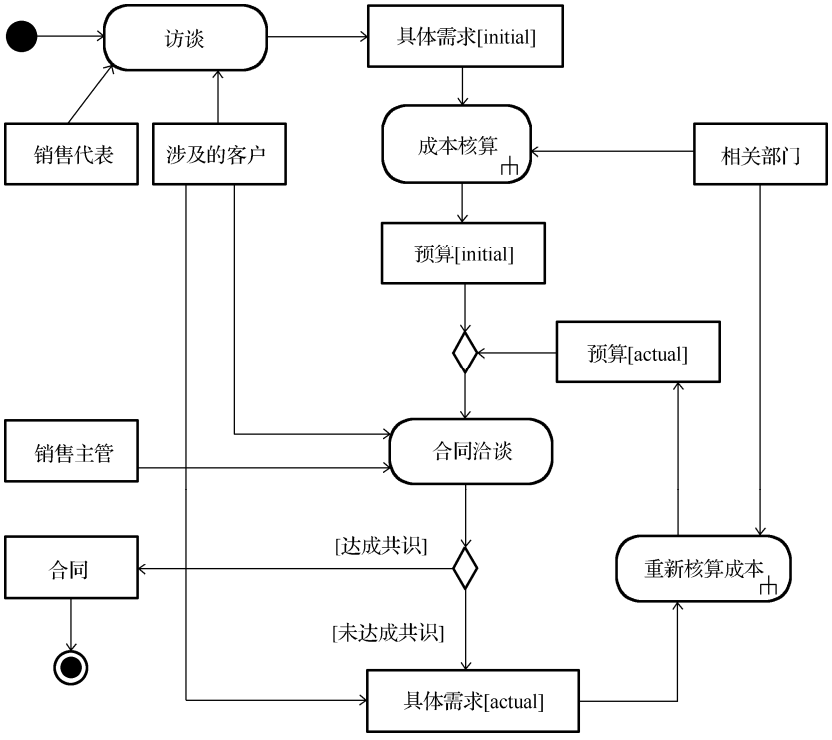


图 3.10　补充细化后的活动图

当所有的可能流程都在模型中体现后，接下来考虑对模型的提炼和优化。图 3.9 和图 3.10 中的动作“成本核算”使用了一个类似“Ψ”的符号标记，表示此动作是一个复合动作，可以通过另外一个活动图进行细化。这个细化的活动图也是按照前面的步骤进行绘制的，首先绘制基本的流程，然后补充可能的流程，最终的结果表示在图 3.11 中，这里去掉了参与者的角色说明，因为按照上层的活动图可知，这里的所有动作都应由相关部门负责。

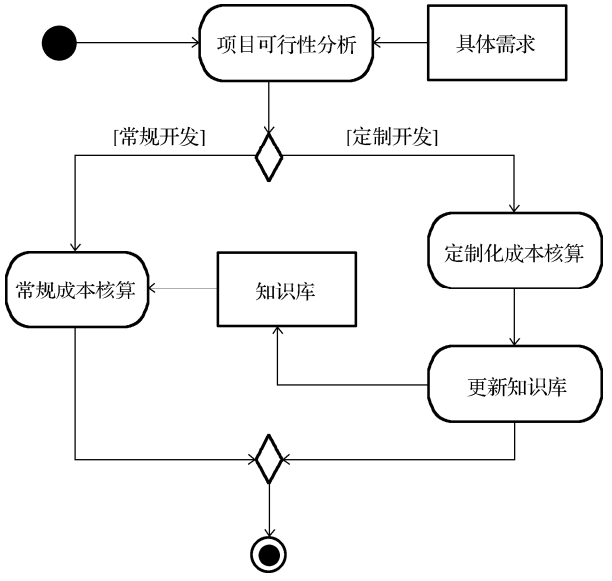


图 3.11　成本核算的细化

在活动图中加入新的业务分支时，通常需要与原有的分支合并，这就会伴随着补充一个菱形的控制，随着这样的控制增多，活动图逐渐变得凌乱了。其实，可以使用一种简化的形式，如图 3.12 所示，省略了对应的菱形控制。但是，活动图的语法规定，若动作具有多个汇聚的箭头，则需要等待所有的分支都完成，类似并行分支的同步情况。因此，在上面的例子中，分别对两个成本的核算对象加入两个状态条件[initial]和[actual]来避免相互等待。

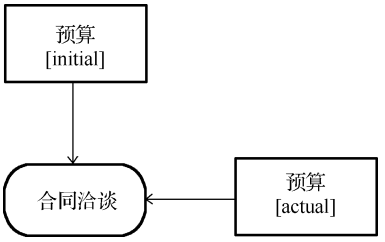


图 3.12　汇聚的另外一种表达

活动图中还有“泳道(Swimlane)”的概念。泳道是一种角色的划分方法，每个角色的活动放置在对应的泳道中。利用泳道可以将模型中的活动按照职责组织起来。在活动图中，这种分配可以通过将活动组织成用线分开的不同区域来表示。例如，图 3.13 中描述的就是某编辑部稿件处理流程对应的活动图，并使用泳道对活动进行了组织。活动图中的泳道可以对应不同的用户角色、用例或者部门等，UML 规范中并没有严格的规定。

以上的例子说明，相同的语法含义在活动图中会有着多种不同的描述方式，本书不对 UML 语法进行面面俱到的介绍，关于活动图的其他描述内容在此不再赘述。

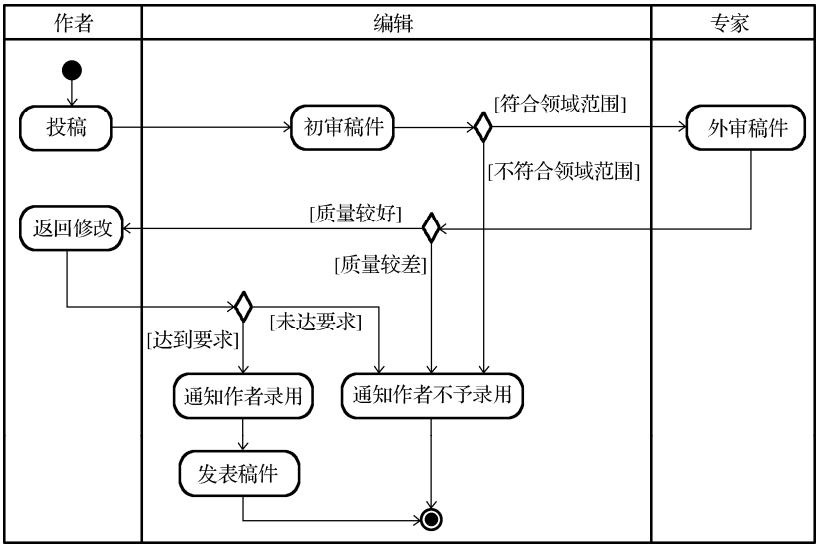


图 3.13　带有泳道的稿件处理流程活动图

3.3.2　事件流

利用活动图进行的过程建模可以应用于用例中的事件流描述。对于用例中基本流的每个步骤，可生成一个单独的动作(action)。如果这个动作比较复杂，可以对应使用几个动作来对它进行表示，或者使用另外一个单独的活动图对其进行细化。然后迭代补充每个备选流的动作，同时注意每个动作是否需要进一步细化。

“软件项目管理系统”中的用例“项目信息编辑”，其基本流的活动图可以表示为图 3.14 的形式，这里暂时没有必要对其中的动作做进一步的细化。然后考虑此用例的备选流，将备选流说明中对应的动作逐渐补充进该活动图，完成后的完整形式如图 3.15 所示。

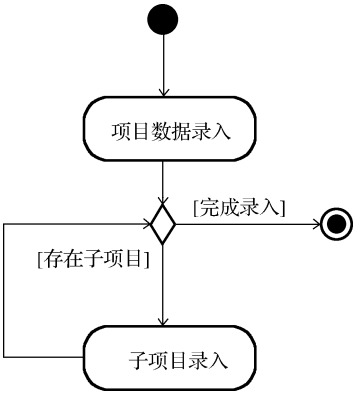


图 3.14　用例“项目信息编辑”的基本流活动图

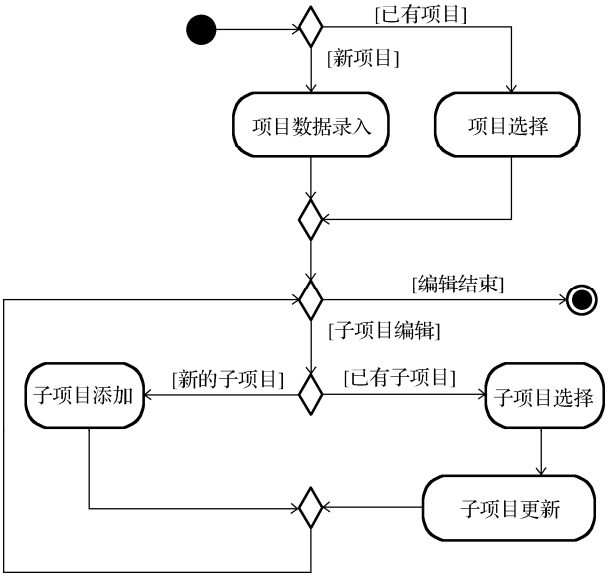


图 3.15　用例“项目信息编辑”的完整活动图

类似地，图 3.16 是根据用例“员工数据维护”的用例规约给出的活动图。在开发的每个阶段，开发者都要考虑对开发结果的可验证性。活动图是一个建立系统测试或验收测试的很好的开始，活动图中每个可能的遍历都对应一种需要进行测试的业务场景。也就是说，我们要对活动图中的每个可能的分支设计一个对应的测试用例，以确保所有的业务情况都能够覆盖到。

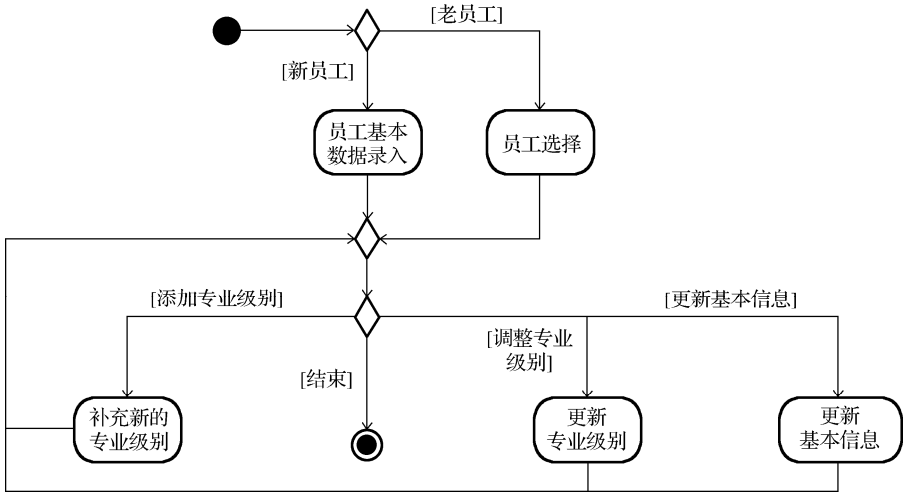


图 3.16 “员工数据维护”用例的活动图

活动图基本都可以在需要进行过程建模的场合中使用，它描述的业务粒度可大可小，如 3.3.1 节中描述的业务就是在一个较高的业务层次上，这时活动图中的动作通常是跨用例的。

3.4 功能性需求

在实际开发中，要优先考虑那些与核心功能相关的活动图，并对它们进行细化和文档化，从用例出发，理解和导出目标软件的具体功能性需求，是后续开发的基础。从用例中抽取功能性需求需要首先识别用例，然后对用例使用模板进行规约并文档化，使用活动图对基本流和备选流进行细化，最后导出文字表述的功能性需求。

一般来说，后续的步骤要比前面的步骤花费更多的时间，各个步骤的工作对最后的需求分析质量都会产生影响，最终的分析结果依赖于业务的复杂程度，以及对它们的理解和细化程度。在分析的过程中，开发者需要从文字描述的功能性需求中提取出分析类图，它描述了业务的结构，也是从业务迈向技术的第一步和关键一步。

对于缺少实际经验的分析人员来说，需求描述本身就是一项较困难的任务。在分析人员与业务客户在一起进行访谈的过程中，业务人员尝试着将他的期望如实地解释出来，分析人员则在他的经验基础上尝试着去理解。同样，开发者在给领域专家解释新的软件系统如何发挥作用时，双方在语言理解方面的误解也会表现出来。由于开发者缺少业务背景，而客户对实现技术又不了解，因此双方的沟通变得障碍重重，主要表现在如下的几个方面。

1. 隐含的假设

当领域专家向开发者说明他们的想法时，一般会将他们认为是理所当然的信息遗漏掉，

谈话也会变得更加简洁。例如，两位开发者正在讨论编程语言 C++和 Java 中对于继承的不同实现方式，这时若另外的人员要无障碍地加入讨论，则他必须要懂得谈话中提到的继承的概念。

2. 笼统的注释

在业务过程描述时，由于存在隐含假设，因此注释就用来对不熟悉的内容进行解释和说明，这也是数据字典的作用。如果业务概念缺失或者过于笼统，就会给开发者带来迷惑。例如，文字说明“数据复杂并且相互关联”，对开发人员来说更容易理解，因为这在技术上很容易与各种数据结构及其表示联系起来，但对一个非专业人员来说，可能就会疑惑：数据到什么程度算是复杂的，没有准确说明的关联数据又有哪些。

3. 模糊的概括

概括是指在基本流中经常使用的“总是”或“从不”等描述形式。分析人员应该在写下需求的同时经常质疑这些描述形式的表达。例如，在需求描述“程序代码改动后类总是会被重新编译”的叙述中，应该搞清楚是否存在例外，是否编译只是在程序代码改动后发生，是否所有的类都必须要被重新编译。

4. 迷惑的命名

在对业务活动进行描述时，不可避免地要对动词进行选择。而名词的使用可以局限在业务层面，数据字典也可以提供解释，这些业务实体在过程的上下文表达中含义通常是清晰的。动词作为名词使用时，要特别注意其表达的含义是否准确。例如，“回归测试要在系统生成后进行”，这里的回归测试和系统生成应该给予说明。

文字形式的需求陈述的目的是从用例中提炼完整而且准确的系统功能性需求。为了能够顺利地进行陈述，可以采用一些规则来辅助生成，并同时保证这些需求的可验证性。建议建立一个文字需求的模板，这是基于语法结构和规则的方法，如图 3.17 所示。

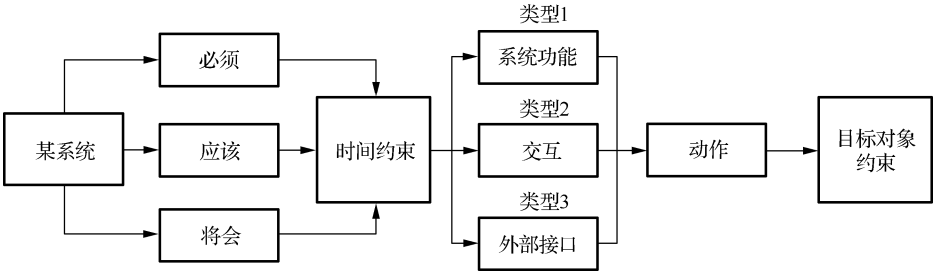


图 3.17　文字需求的模板

图 3.17 中给出了文字需求的模板中的三种基本形式：通过“必须”“应该”“将会”控制右侧需求陈述的组合。这里，“必须”的需求有着重要作用，因为其代表用户直接的期望；“应该”的需求意味着从用户的角度出发，其可能是有意义的，但是否需要最终进行实现还可继续讨论；“将会”的需求通常表示在某些已经实现的功能上的扩展功能，需要考虑它们的可行性。

从合同的角度看，开发者(乙方)当然希望把合同框架只限定在“必须”的需求范围之内。但这在实践中往往不是令客户(甲方)满意的做法。对于开发者来说，总是希望能与客户保持较好的、长期的合作关系。

模板中还存在三种不同的需求类型，如图 3.18 所示，它们的具体含义如下：

(1)类型 1：系统功能，即系统自身应完成的功能需求，例如，根据所有子项目及其完成的进度，查询和计算项目到目前为止已经投入的工作量。

(2)类型 2：交互，即该系统提供给特定用户交互性的功能，例如，在进行项目数据录入时，系统需要提供给用户对项目包含的相关字段进行录入的界面。

(3)类型 3：外部接口，即该系统为第三方系统提供的外部访问接口或者需要通过外部系统的接口获取数据的功能。例如，“软件项目管理系统”为外部的办公系统提供查询接口，提供进行中的项目数据浏览和数据交换等功能。

类型 1 是系统功能的主要组成部分，同时也是类型 2 和类型 3 的发起者。类型 2 一般提供交互类的界面，然后执行对深层功能的调用并显示结果。这些深层功能一般为类型 1 的功能。

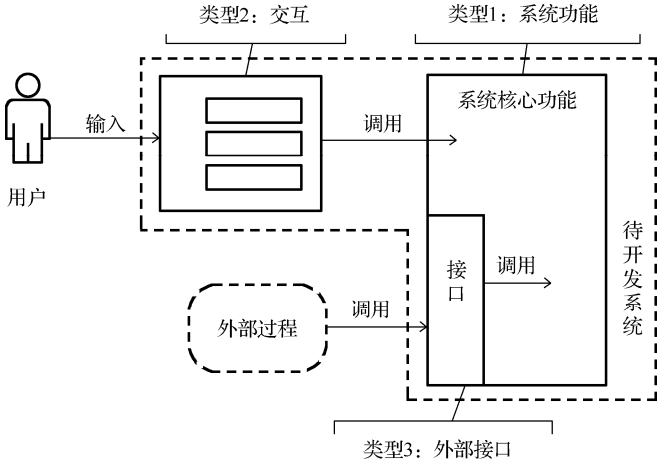


图 3.18　三种需求类型间的关系

模板中的时间约束指示具体的约束条件，即对应的功能必须满足的要求。在一些基于交互的系统中，开发者经常会通过以下形式对需求进行描述：“当用户选择功能 A 后，系统必须……”“当输入的 E 通过有效性检查后，系统要……”。

所有的需求描述，都具有目标对象约束，尽可能补充那些缺少的或模糊的附加内容。需求的文字描述从用例的活动图出发，产生并最终形成对未来系统的功能需求。在需求陈述的文档中，要对需求进行统一的编号并赋予简短的标题，用以跟踪该需求的实现。作为说明，从用例“项目信息编辑”(图 3.15)中可以得出以下的功能性需求：

(1)R1.1 项目创建：系统必须提供给用户项目创建及为其指定具体项目信息的功能。

- 项目信息：自动生成的唯一项目编号、项目名称、项目起止时间、预计工作量。
- 项目：可由多个子项目构成，同时，子项目也可作为单独的项目。除子项目外，项目还可分解为具体的任务。
- 子项目：项目可由子项目构成，从结构上看，二者是相同的。

(2)R1.2 数据存储：数据输入完成后(按回车键或单击“确认”按钮)，系统必须将新输入的记录在系统中持久地存储起来。

(3)R1.3 项目选择：系统必须提供给用户选择某个项目的功能。

(4)R1.4 子项目创建：选定项目后，系统需要提供给用户为所选项目创建子项目的功能。

(5)R1.5 子项目与项目：系统对子项目的处理方式，与项目应该是一样的，对项目提供的编辑功能，子项目也必须具有。

(6)R1.6 项目信息编辑：选定项目后，系统必须提供给用户对该项目进行数据编辑的功能，包括实际开始时间、最新计算出的结束时间、预计工作量及项目备注等。

(7)R1.7 项目任务添加：选定项目后，系统必须提供给用户对该项目添加项目任务的功能。

- 项目任务：项目中包含的子任务，具有名称、责任人、计划与实际工作量、计划与实际的开始和结束时间、完成进度等属性。
- 完成进度：每次数据更新后，对项目任务的完成进度通过百分数进行标识，通常是线性递增的。项目的完成进度是根据其子项目及任务的进度，通过预计工作量值计算出来的。
- 工作量：是指每次编辑项目任务记录时，为此任务指定的已花费时间(小时)。整个项目的工作量根据每个子项目和任务的工作量进行计算。

(8)R1.8 项目任务选择：项目选定后，系统必须提供给用户对项目任务进行选择的功能。

(9)R1.9 项目任务编辑：选定项目任务后，系统必须提供给用户对选定任务的所有属性进行修改的功能。

(10)R1.10 对其他项目的依赖：选定项目后，系统必须提供给用户对该项目与其他(子)项目的依赖关系的编辑功能，说明它们之间的逻辑关系。

(11)R1.11 工作量改动的验证：录入新的子项目、新的任务或对子项目或任务的工作量改动后，系统必须对工作量值的合理性进行检查。

(12)R1.12 工作量检查：在系统进行工作量合理性检查时，必须保证所有子项目和任务的计划工作时间之和小于等于所属项目的总计划工作时间。

(13)R1.13 工作量改动失败提醒：如果某项工作时间的改动没有通过合理性检查，系统就必须通知用户存在的问题，而且不允许更改对应的工作时间属性。

需求 R1.2 是一个通用的功能，因为可能还会同时给其他活动图中的业务提供存储服务，这些依赖关系可以通过需求跟踪来指定。进一步，对于交叉使用的功能，可以使用单独的编号方式进行区分。例如，S1.1 为一个交叉使用的需求描述——系统功能选择。

S1.1 功能选择：系统启动后，系统提供给用户对以下功能进行选择的界面：项目和任务编辑、开发者管理和项目监控。

这个例子中没有涉及类型 3 外部接口，这是因为该系统是一个独立的系统，并没有涉及必须与其他外部系统的交互。若用户要求该系统为外部系统提供项目一览的列表，则对应的需求可描述如下。

R1.14 与外部系统的协同：通过与外部系统取得连接后，系统必须能够对外部系统的查询请求(如项目名称、工作时间汇总及完成进度等)结果进行返回。

由活动图派生的文本形式的需求，需要将每个动作都在一个或多个需求陈述中体现，而且每个动作的转移和判断分支，都至少要包含在一个功能中，或者作为约束条件在需求中存在，不要遗漏。

3.5 非功能性需求

到目前为止，讨论的都是功能性需求，功能性需求的实现保证了软件系统构成的完整性，能够为用户提供成功的应用。除功能性需求外，用户可能还会提出另外一些关于系统可用性方

面的需求，例如，系统的响应时间，它使用户能够有效地使用软件工作，不允许存在较大的延迟。在需求调研时，还存在一些系统涉众，对于他们来说，没有纯功能上的需求，而是要满足他们对软件产品某些方面的要求，如可移植性或者文档的完整性等。所有这类需求统称为非功能性需求。

1. 质量需求

质量需求是针对目标软件的质量特征进行说明的。有关质量方面的内容在表 3.4 中进行了概括，类似于 ISO 9126 中定义的质量框架标准。当然，也可以将功能性需求也归并到质量需求中，因为对于软件的正确性而言，能够满足功能上的需求也是质量的一部分。

表 3.4　软件产品的质量需求

质量特征	质量特征的具体定义和示例	可能的质量措施
正确性	符合规格说明的程度 例如，软件根据算法 A 计算速度	根据算法 A 制订对应的测试用例
安全性	系统的人身伤害或财产损失限制在可接受的水平之内是安全的 例如，对于数据 XY 的访问受到密码保护	在系统空闲时间，可设置一个暴力破解程序，尝试所有可能的密码组合，对系统进行尝试性攻击
可靠性	在给定的环境下、固定的时间间隔中，系统没有出现故障的概率 例如，系统应能保证 24 小时×7 天正常运行	系统平均无故障运行时间(MTTF)：系统启动后正常运行的平均时间
可用性	对系统功能性的一种度量，指系统在某时间点无故障运行的概率 例如，系统在每天早上 7:30 能够正常运行	平均无故障运行时间(MTTF)加上平均维修时间(MTTR)：系统维修和重启的平均时间
健壮性	系统在非常规环境中工作的能力及对异常的容错能力 例如，当系统出现故障时，软件能够自动从主服务器切换到备用服务器，而对用户的使用没有影响	统计在电源断电后的情况下，系统能够成功正常运行的次数
存储和运行效率	针对不同的负载情况，系统的存储消耗及运行状况 例如，系统在 PC 机上占用的内存上限为 500MB	使用内存监控软件测试在不同负载下占用的内存量
可维护性	在操作条件或需求变更的条件下对软件进行调整，以适应用户的期望 例如，系统的架构清晰，修改容易	开发过程中要不断地进行重构，采用必要的设计模式辅助设计
可移植性	软件系统迁移到其他平台上运行的能力，一般与用户的需求相关 例如，系统后台数据库从 SQL Server 迁移到 Oracle	了解两种数据库的差异，并在设计和实现中屏蔽偏差
可验证性	对于程序的正确性、健壮性及可靠性进行测试的可能性 例如，系统生成 1000 页文档的时间小于等于 300 毫秒	开发过程中采用措施提升系统的可测试性
易用性	软件系统对用户应友好，并且具有简单、易用的特点 例如，用户应易于学习和掌握系统的操作	对用户输入的提示、标准格式的约束等

在对具体的需求进行描述时要注意，对给出的需求陈述应能满足，并且程度是可以度量的，为此要尽可能准确和详细地给出可能采取的质量验证措施。

在实际开发中，质量需求尤其要引起开发者的关注，因为某个单一的质量需求可能会对整个项目的成败起到决定性的作用，而单一的功能性需求的影响力往往没有这么大，不满足功能性需求往往只是不满足系统的某些功能而已，通常不会危及整个项目。例如，某非功能性需求要求开发的系统在计算结果展示时，5s 内必须返回结果，如果逾越这个限制，对用户来说，整个系统可能都是失败和无法接受的。

此外，一个简短的质量需求后面可能会存在着大量的隐性需求。例如，要符合某个标准或规范的要求，意味着该标准或规范涉及的所有文档内容都会成为需求的一部分。

2. 技术性需求

技术性需求主要是指直接针对软件项目相关的技术及环境等方面的需求，其中最典型的是硬件需求。例如，需要描述清楚支持软件系统运行的硬件平台。除此之外，还有支持的操作系统及与未来软件共同运行的其他软件要求等，例如它们的版本等。功能性需求中主要阐述与协同软件的协作功能，而技术性需求主要包含对协同软件的约束，包括版本、分布式架构、连接类型、网络传输速度等，它们对未来软件的性能也会产生一定的影响。

技术性需求还可以描述为对开发环境或工具的要求，如将开发工具指定为 Java 的某个版本或者函数库等。总之，需要保证新的软件能够在要求的环境中运行。如果客户期望以后自己能够承担维护或修改未来软件的工作，使用的开发环境也会作为技术性需求的一部分。

3. 其他交付物

软件项目通常并不只是交付最终的安装包，其他与产品相关的交付物，如硬件、使用说明书、安装说明书、依赖库、开发文档及培训等，都需要作为非功能性需求列出。

4. 合同需求

其他边界和约束条件，如项目开发的进度安排等，同样也是需求的一部分。这些需求的形成可以通过合同的形式规范下来，或者作为补充的合同条款细化。这是合理的，因为付款的方式、交付期限、计划的项目会议、惩罚和诉讼方式等，对未来开发都有着影响和牵动作用。

5. 规格说明

所有的需求最终都要在需求文档中进行汇集，这份文档通常称为需求规格说明书。合同的甲方，即委托方，在说明书中明确对未来产品的期望；合同的乙方，即开发方，将要完成的需求任务通过说明书确定下来。由此，双方在需求层面上达成共识。所以需求规格说明书不仅仅是一份合同。通常所说的商务合同一般只是一份较为粗略的任务说明，对于合同的甲方和乙方来说，可能还需要一份更为详细的技术协议，作为商务合同的细化和补充。需求规格说明书的结构如下。

(1) 文档说明性内容：文档的版本号、创建者、修改记录、批准者等。

(2) 目标群体和系统目标：利益相关者分析、项目目标、预算成本、开发周期等。

(3) 功能性需求：从用户故事到业务场景分析，再到文字的功能性陈述。

(4) 非功能性需求：注重质量需求及重要的技术需求。

(5) 其他交付物：需要交付的软件制品(Artifact)清单及时限。

(6) 验收标准：验收的方式及标准说明。

(7) 附件：相关文档的清单。

3.6 需求跟踪

项目的开发始于需求，需求的质量也直接决定了未来项目开发的成败。需求管理同样是项目管理的最佳实践之一，尤其对于大型项目或者需要长期扩展开发的项目，将需求管理起来并且能够随时获知需求的完成情况，对项目的开发工作具有很大的帮助。

需求跟踪的基本原理是给出每个原始需求项及其对应的目标制品，并将它们通过某种方

式联系起来，由此可以确定出源需求的实现情况。对类似的“源需求项→目标制品”的映射都应该通过跟踪进行管理，如“用例图→活动图”“活动图或者动作(Action)→功能性需求”“功能性需求→类”“功能性需求→测试用例”等。

对这些关联信息的管理会使得跟踪变得方便，最简单的方法是使用一个文档快速地把这些对应关系记录下来。但是，如果跟踪的内容有变化，需要对该文档中的大量关联内容进行维护，这是非常烦琐的，因此引入工具的支持是非常必要的。在某些 UML 工具中已经紧密地集成了对这些关联的管理功能，例如，对于用例图中的用例，可以指定其与活动图之间的连接，通过这些连接可以从用例直接导航到对应的活动图。

图 3.19 粗略地描述了这样的工具的跟踪关系，这样的跟踪关系有时可能比较复杂，因为源与目标之间经常存在多对多的联系。

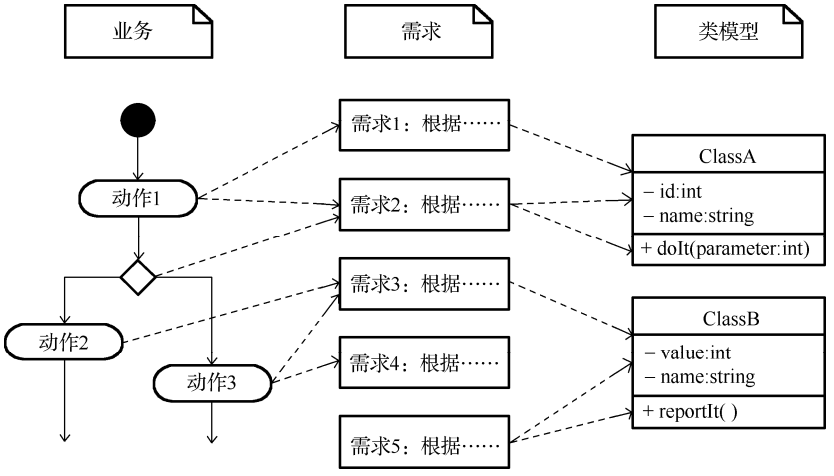


图 3.19　跟踪关系图

使用图 3.19 的跟踪信息进行分析的一个直接的应用就是能够马上识别出一个需求有没有源内容与其对应。例如，图中的需求 5 就需要进一步确认是否遗忘了与它对应的源或者需求 5 确实不存在源，在这种情况下，预示了该需求描述并非由客户提出，因此有可能会将此需求省略，不做实现。类似地，图中的需求 4 没有对应的目标实现。

跟踪信息另外一个非常有价值的地方在于，如果软件随后进行了修改或者扩展，在正式实施前就存在一个问题，那就是为了需求变更的有效实现，需要事先对其影响范围进行评估。根据跟踪的对应连接，可以很容易地回答这些问题。同样，如果软件的某个功能被删除了或者进行了调整，通过跟踪信息马上也会意识到有哪些类受此影响，进而可以逆向追溯需求，以获知影响的功能范围。跟踪信息还可以附加创建时间、迭代周期等信息，以满足更多的追踪需求。

在能力成熟度模型(Capability Maturity Model for Software，CMM)中，要求软件组织必须具备需求跟踪的能力，即“在软件制品之间，维护一致性”。软件制品包括软件计划、过程描述、分配需求、软件需求、软件设计、代码、测试计划及测试过程。由此可见，需求追踪活动是覆盖整个开发周期的，例如，建立测试用例和需求之间的连接和跟踪，对提高测试的完备性具有重要意义。

在实践中，我们常常借助需求跟踪矩阵(Requirement Tracking Matrix，RTM)对变更进行

跟踪，包括需求变更、设计变更、代码变更、测试用例变更等。需求跟踪矩阵是目前经过实践检验的进行变更波及范围影响分析的有效工具，如果不借助 RTM，那么发生上述变更时，往往会遗漏某些连锁变化。RTM 是验证需求是否得到实现的有效工具。

按照跟踪的内容，可以将 RTM 分为纵向跟踪矩阵和横向跟踪矩阵两类。纵向跟踪矩阵又包括如下三种跟踪内容：

(1) 需求之间的派生关系，如从客户需求到系统需求。

(2) 实现与验证的关系，如从需求到设计、从需求到测试用例等。

(3) 需求的责任分配关系，如需求的负责人员。

横向跟踪矩阵主要包含的内容有需求之间的接口关系等。

RTM 并没有规定具体的实现办法，每个团队注重的方面不同，所创建的需求跟踪矩阵也不同，只要能够保证需求链的一致性和状态的跟踪，就能达到目的。

3.7 习题

1．在软件开发过程中，问题发现得越晚，修正起来越困难，付出的代价越高。分析原因，至少给出两个理由，并简短说明。

2．图 3.10 通过活动图描述了合同签订的流程，请对其进行扩展，满足以下要求。

(1) 客户对销售的产品并不感兴趣。

(2) 对于金额小于 20 万元的合同，在谈判的过程中需要部门经理的参与，考虑到项目管理的费用，由他们决定是否按照当前的合同金额签订合同。

(3) 相关部门在成本核算前会提出一些问题，需要销售人员向客户询问，并澄清结果。

3．给出以下需求描述的用例图。

(1) 一个音像商店准备开发软件系统，用于向客户销售或者租借电影光盘。

(2) 音像商店向多家订购商订购光盘，然后分类存储在系统中，订购了上千张光盘；还可以根据客户的请求向订购商订购光盘。

(3) 所有的电影光盘用条码来管理，条码的号码是光盘的唯一标识。

(4) 音像商店可以向客户销售或租赁电影光盘。使用条码扫描来支持销售或租赁。

(5) 音像商店建立会员制，会员客户购买电影光盘可以享受折扣。会员卡也使用条码来管理。

(6) 会员可以通过网络预订电影光盘，并在指定的日期来取。

(7) 会员可以利用灵活的搜索机制找到喜欢的电影，如果没有对应光盘，可以提出预订。

使用 UML 给出上述需求描述的用例图。要求绘制规范，尤其注意“角色—用例”和“用例—用例”之间的关系。

4．根据某毕业设计选题系统的功能描述，使用 UML 建模技术，完成需求分析的用例图，包括系统的用例及其子用例(如果有，需要标记与主用例的关系)和角色(Actor)。

(1) 教师信息维护：教务员录入老师的基本信息；教师信息包括教师 ID、教师姓名、教师职称、联系方式、邮箱地址等，可从 Excel 中导入；指导教师的联系方式在学生选题申报成功后，才能公开给学生。

(2) 学生信息维护：教务员录入和维护学生信息，学生信息包括学号(学生 ID)、学生姓名、班级。

(3) 登录：学生、教师、教务员都需要输入 ID 和密码登录系统，使用权限范围内的用例，可以修改个人密码。

(4) 出题：教师使用此功能登记和维护毕业设计的题目。子功能是在出题过程中，要确定题目的类型，如校内或者校外，可选的是直接指定该题目的选题学生。

(5) 审题：系主任负责对所有该系教师出的题目进行审核，合格的题目可以发布，不合格的题目要求教师修改。

(6) 开放选题：教务员将所有审核通过的题目公开，供学生选择。

(7) 选题：学生浏览公开的题目列表，根据题目要求和个人兴趣及特点，选择相应的题目。

(8) 确认选题：教师审查自己所出题目的选题情况，对合格的学生予以确认，将不合格的学生删除，并发送邮件通知。

(9) 统计选题情况：教务员在选题结束后统计选题情况，包括已选和未选的情况。

5．以下给出了“老年人监护系统”中的用例及其描述，使用 UML 用例图描述该系统，并给出用例之间的联系。

(1) 摔倒动作检测：从楼梯传感器和摄像头中获取输入数据，用以检测是否有人摔倒。

(2) 摔倒事件报警：如果检测到某位老人摔倒，那么发送一条报警消息到手机上，同时该报警信息会被发送到用例“事件日志”中进行记录。

(3) 事件日志：将发生的事件记录在数据库中。

(4) 床传感器监测：从安装在床位上的床传感器中获取脉搏、呼吸等数据，并发送到用例“事件日志”中处理。

(5) 配置系统：系统管理员对系统进行各种配置操作。

6．在图书管理系统中：

(1) 管理员可进行“删除书籍”和“修改书籍信息”这两个操作，并且这两个操作在执行前都必须先进行“查询书籍”操作。

(2) 读者可以“还书”，这是一个基础操作。如果读者所借书籍超期，还书时要缴纳罚金，即当书籍“超期”时，将执行“缴纳罚金”操作。

要求：画出上述系统的 UML 用例图。

第 4 章　软件架构的构建

软件架构也称软件体系结构。在对软件工程的初期研究中，人们把软件设计的重点放在数据结构和算法的设计和选择上，随着软件系统的规模和复杂程度不断增加，总体的系统结构设计和规格说明比起计算的算法和数据结构已经变得重要得多。在此种背景下，人们认识到软件架构(Software Architecture)的重要性，并认为对软件架构的系统、深入的研究将会成为提高软件生产效率和解决软件维护问题的新的、最有希望的途径。

用户需求中的各种约束对软件架构的选择都有直接的影响。本章主要介绍软件架构的基本概念、相关模型、风格及设计方法。

4.1　软件架构及其定义

4.1.1　软件架构的理解

软件架构设计就是建立计算机软件系统所需的数据结构和程序构件，主要考虑软件系统采用的体系结构风格、系统组成构件的结构和属性及系统中所有体系结构构件之间的相互关系。

可以把软件比喻为一座楼房，从整体上看，它有基础、主体和装饰，例如操作系统之上的基础设施软件、实现计算逻辑的主体应用程序和方便使用的用户界面程序。另外，从细节上看，每个程序也是有结构的。早期的结构化程序就是以语句组成模块，模块的聚集和嵌套形成层层调用的程序结构，也就是体系结构。结构化程序的程序(表达)结构和(计算)逻辑结构的一致性及自顶向下的开发方法自然而然地形成了体系结构。由于结构化程序设计时代程序规模不大，通过强调和应用结构化程序设计方法学，自顶向下、逐步求精，并注意模块的耦合性，就可以得到相对良好的结构，因此，并未特别研究软件架构。

还是用建筑来比喻，结构化程序设计时代是以砖、瓦、灰、沙、石来预制梁、柱、屋面板盖平房和小楼，而面向对象时代以整面墙、整间房、一层楼梯的预制件来盖高楼大厦。构件怎样搭配才合理？体系结构怎样构造更容易？重要构件有了更改后，如何保证整栋高楼不倒？每种应用领域(医院、工厂、旅馆等)需要什么构件？有哪些实用、美观、强度、造价合理的构件骨架使建造出来的建筑(即体系结构)更能满足用户的需求？如同土木工程进入现代建筑学一样，软件也从传统的软件工程进入现代面向对象的软件工程，研究整个软件系统的体系结构，并寻求构造最快、成本最低、质量最好的构造过程。

软件架构研究的主要内容涉及软件架构描述、软件架构风格、软件架构评价和软件架构的形式化方法等，目的是解决好软件的重用、质量和维护问题。综观软件架构技术的发展过程，从最初的“无结构”设计到现行的基于体系结构的软件开发，共经历了 4 个阶段：

(1)无体系结构设计阶段。以汇编语言进行小规模应用程序开发为特征。

(2)萌芽阶段。出现了程序结构设计主题，以控制流图和数据流图构成的软件结构为特征。

(3) 初期阶段。出现了从不同侧面描述系统的结构模型，以 UML 为典型代表。

(4) 高级阶段。以描述系统的高层抽象结构为中心，不关心具体的建模细节，划分了体系结构模型与传统软件结构的界限，该阶段以 Kruchten 提出的“4+1”模型为标志。

4.1.2 软件架构的定义

虽然软件架构已经在软件工程领域有广泛应用，但迄今为止还没有一个被大家所公认的定义。许多专家学者从不同角度对软件架构进行了刻画，较典型的定义有以下几个。

(1) Dewayne Perry 和 Alexander Wolf 认为，软件架构是具有一定形式的结构化元素(Element)，即构件的集合，包括处理构件、数据构件和连接构件。处理构件负责对数据进行加工，数据构件是被加工的信息，连接构件把体系结构的不同部分组合、连接起来。这个定义注重区分处理构件、数据构件和连接构件，并在其他的定义和方法中基本得到了保持。

(2) Mary Shaw 和 David Garlan 认为，软件架构是软件设计过程中的一个层次，这一层次超越计算过程中的算法和数据结构设计，并在其上处理整体系统结构设计和描述方面的一些问题，如全局组织和全局控制结构、通信、同步与数据存取的协议(Protocol)、设计构件功能定义、物理分布与合成、设计方案的选择、评估(Evaluation)与实现等。

(3) Kruchten 指出，软件架构有四个角度，它们从不同方面对系统进行描述。概念(Concept)角度描述系统的主要构件及它们之间的关系；模块角度包含功能分解与层次结构；运行角度描述一个系统的动态结构；代码角度描述各种代码和库函数在开发环境中的组织。

(4) Barry Boehm 和他的学生提出，软件架构包括软件和系统构件、互联及约束的集合；系统需求说明的集合；基本原理用以说明这一构件、互联和约束能够满足系统需求。

(5) Bass、Ctements 和 Kazman 认为，程序或计算机系统的软件架构包括一个或一组软件构件、软件构件的外部可见属性及其相互关系。其中，软件构件的外部可见属性是指软件构件提供的服务、性能、特性、错误处理、共享资源使用等。

综上所述，软件架构的通识定义可以描述为：软件架构为软件系统提供了一个结构、行为和属性的高级抽象，由构成系统的元素的描述、这些元素的相互作用、指导元素集成的模式及这些模式的约束组成。软件架构不仅指定了系统的组织(Organization)结构和拓扑(Topology)结构，并且显示了系统需求和构成系统的元素之间的对应关系，提供了一些设计决策的基本原理。

4.1.3 软件架构的“4+1”视图模型

研究软件架构的首要问题是如何表示软件架构，即如何对软件架构建模。根据建模的侧重点不同，可以将软件架构的模型分为五种：结构模型、框架模型、动态模型、过程模型和功能模型。在这五种模型中，最常用的是结构模型和动态模型。

(1) 结构模型。这是一个最直观、最普遍的模型。这种方法以体系结构的构件(Component)、连接件(Connector)和其他概念来刻画结构，并力图通过结构反映系统的重要语义内容，包括系统的配置、约束、隐含的假设条件、风格、性质等。研究结构模型的核心是体系结构描述语言。

(2)框架模型。框架模型与结构模型类似，但它不太侧重描述结构的细节，而更侧重整体结构。框架模型主要以一些特殊的问题为目标，建立只针对和适应该问题的结构。

(3)动态模型。动态模型是对结构或框架模型的补充，研究系统“大颗粒”的行为性质。例如，描述系统的重新配置或演化。动态可以指系统总体结构的配置、建立、注销通信通道或计算的过程。这类系统通常是激励型的。

(4)过程模型。过程模型研究构造系统的步骤和过程，因而其结构遵循某些过程脚本的形式。

(5)功能模型。功能模型认为体系结构由一组功能构件按层次组成，下层向上层提供服务。它可以视为一种特殊的框架模型。

软件架构的五种模型各有所长，因此形成一个完整的模型来刻画软件架构更为合适，并将五种模型有机地统一。Kruchten 在 1995 年提出了一个“4+1”的视图模型。“4+1”视图模型用五个不同的视图(逻辑视图、进程视图、物理视图、开发视图和场景视图)来描述软件架构。每个视图只关心系统的一个侧面，五个视图结合在一起才能反映系统的软件架构的全部内容。“4+1”视图模型如图 4.1 所示。

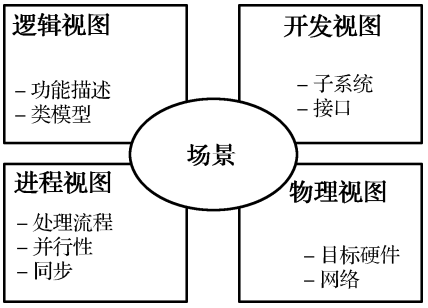


图 4.1 “4+1”视图

逻辑视图(Logic View)主要支持系统的功能性需求，即系统提供给最终用户的服务。在逻辑视图中，系统被分解成一系列的功能抽象，这些抽象主要来自问题领域。这种分解不但可以用来进行功能分析，而且可用来标识在整个系统的各个不同部分的通用机制和设计元素。在面向对象技术中，通过抽象、封装和继承，可以用对象模型代表逻辑视图，用类图(Class Diagram)描述逻辑视图。

开发视图(Development View)也称模块视图(Module View)，主要侧重于软件模块的组织和管理。软件可通过程序库或子系统进行组织，这样，一个软件系统，就可以由不同的人进行开发。开发视图要考虑软件内部的需求，如软件开发的容易性、软件的重用性和软件的通用性，要充分考虑由于具体开发工具的不同而带来的局限性。

进程视图(Process View)也称并发视图，侧重于系统的运行特性，主要关注非功能性需求，例如，系统的性能和可用性。进程视图强调并发性、分布性、系统集成性和容错能力，以及逻辑视图中的主要抽象如何适合进程视图。它也定义逻辑视图中的各个类的操作具体是在哪一个线程(Thread)中被执行的。

物理视图(Physical View)主要考虑如何把软件映射到硬件上，它通常要考虑系统性能、规模、可靠性等，解决系统拓扑结构、系统安装、通信等问题。当软件运行于不同的节点上时，各视图中的构件都直接或间接地对应系统的不同节点。因此，从软件到节点的映射要有较高的灵活性，当环境改变时，对系统其他视图的影响要最小。

场景视图(Scenarios View)可以视为那些重要系统活动的抽象，它使以上 4 个视图有机联系起来，从某种意义上说，场景是最重要的需求抽象。在开发体系结构时，它可以帮助开发者找到体系结构的构件和它们之间的作用关系。同时，也可以用场景来分析一个特定的视图，或描述不同视图构件间是如何相互作用的。

4.2 软件架构模型

综合软件架构的概念，体系结构的核心模型由五种元素组成：构件、连接件、配置(Configuration)、端口(Port)和角色(Role)。其中构件、连接件和配置是最基本的元素。

(1)构件是具有某种功能的可重用的软件模板单元，表示系统中主要的计算元素和数据存储。构件有两种：复合构件和原子构件，复合构件由其他复合构件和原子构件连接而成；原子构件是不可再分的构件，底层由实现该构件的类组成。这种构件的划分提供了体系结构的分层表示能力，有助于简化体系结构的设计。

(2)连接件表示构件之间的交互，简单的连接件有管道(Pipe)、过程调用(Procedure Call)、事件广播(Event Broadcast)等，更为复杂的交互有客户机/服务器(Client/Server)通信协议、数据库和应用之间的 SQL 连接等。

(3)配置表示构件和连接件的拓扑逻辑和约束。

4.3 软件架构风格

软件架构设计的一个核心问题是能否使用重复的体系结构模式，即能否达到体系结构级的软件重用，或者说能否在不同的软件系统中使用同一体系结构。

软件架构风格是描述某一特定应用领域中系统组织方式的惯用模式。体系结构风格定义了一个系统家族，包括体系结构的定义、词汇表和一组约束。词汇表中包含一些构件和连接件类型，约束指出系统是如何将这些构件和连接件组合起来的。体系结构风格反映了领域中众多系统共有的结构和语义特性，并指导如何将各个模块和子系统有效地组织成一个完整的系统。按这种方式理解，软件架构风格定义了用于描述系统的术语表和一组指导构建系统的规则。

软件架构风格促进了对设计的重用，不变的部分使不同的系统可以共享同一实现代码，只要系统使用常用的、规范的方法组织，就可使其他开发者很容易地理解系统的体系结构。

根据此框架，Garlan 和 Shaw 给出了软件架构风格的分类：

(1)数据流风格：批处理序列、管道与过滤器等。

(2)调用/返回风格：层次结构、正交软件结构、客户机/服务器结构、浏览器/服务器结构等。

(3)独立构件风格：进程通信、事件系统、MVC 结构等。

(4)虚拟机风格：解释器、基于规则的系统等。

(5)数据中心风格：数据库系统、超文本系统、仓库/黑板系统等。

本书介绍几种主要的软件架构风格，它们常用于主流系统的设计。

4.3.1 管道与过滤器

在管道与过滤器风格的软件架构中，每个构件都有一组输入和输出，构件读取输入的数据流，经过内部处理，然后产生输出数据流。这个过程通常通过对输入流的变换及增量计算完成，

所以在输入被完全处理之前，输出便产生了。因此，这里的构件也称过滤器，这种风格的连接件就是数据流传输的管道，将一个过滤器的输出传到另一个过滤器的输入。这里的过滤器必须是独立的实体，它不能与其他过滤器共享数据，而且一个过滤器不知道它上游和下游的标识。图 4.2 是管道与过滤器风格的示意图。

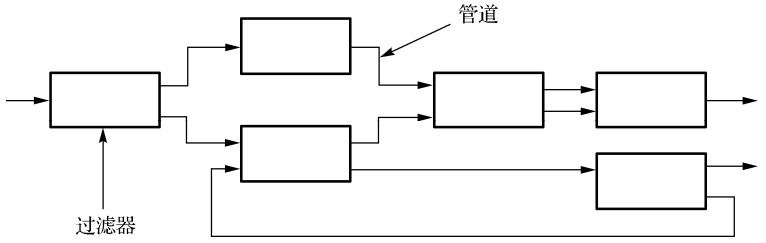


图 4.2　管道与过滤器风格

一个典型的管道与过滤器风格的例子是以 UNIX Shell 编写的程序。UNIX 既提供一种符号，以连接各组成部分(UNIX 的进程)，又提供某种进程运行时的机制以实现管道。另一个著名的例子是传统的编译器，传统的编译器一直被认为是一种具有管道与过滤器风格的系统，在该系统中，一个阶段(包括词法分析、语法分析、语义分析和代码生成)的输出是另一个阶段的输入。

管道与过滤器风格的软件架构通常适合批处理和非交互处理的系统，通过这种软件架构使得软件具有良好的信息隐藏性和模块独立性，从而产生高内聚、低耦合的特点。

4.3.2　层次结构

层次系统组织成了一个层次结构，每一层为上层提供服务，并作为其下层客户。在一些层次系统中，除了一些精心挑选的输出函数，内部的层通常只对相邻的层可见。这样的系统中构件在一些层上实现了虚拟机，而在另外的层次系统中层可能是部分不透明的。连接件通过决定层之间如何交互的协议来定义，拓扑约束包括对相邻层间交互的约束。

这种风格支持可增加抽象层的设计，允许将一个复杂问题分解成一个增量步骤序列实现。由于每一层最多只影响两层，只要给相邻层提供相同的接口，就允许每一层用不同的方法实现，为软件重用提供了强大的支持。

图 4.3 是层次结构风格的示意图。层次结构风格最广泛的应用是分层的网络通信协议，如 ISO 的七层协议栈结构，每一层提供一个抽象的功能，作为上层通信的基础，较低的层次定义低层的交互，最低层通常只定义硬件物理连接。

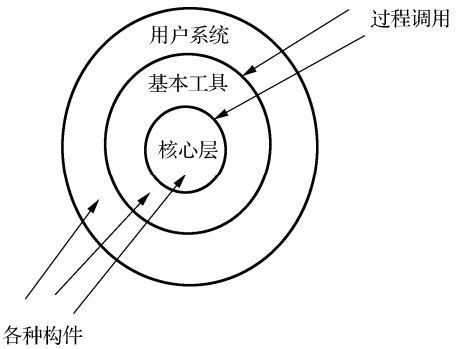


图 4.3　层次结构风格

层次结构是基于抽象程度递增的系统设计，使开发者可以把一个复杂系统按递增的步骤进行分解，而且便于功能的增强和扩展。图 4.4 是 Windows 中的 TCP/IP 层次结构，是一个典型的层次结构风格。

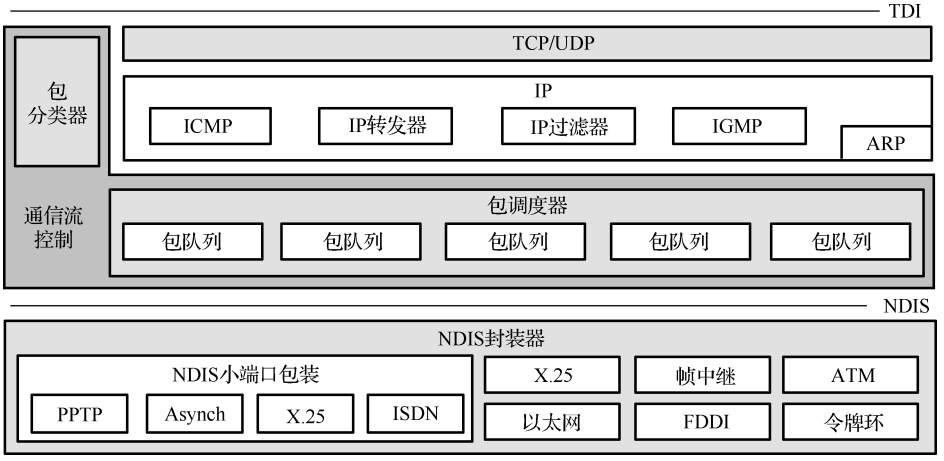


图 4.4　Windows 中的 TCP/IP 层次结构

4.3.3　仓库/黑板系统

在仓库/黑板系统风格中，有两种不同的构件：中央数据结构，说明当前状态；独立构件，在中央数据存储上执行。仓库与外构件之间的相互作用在系统中会有较大的变化。

控制原则的选取将产生两个主要的子类。若输入流中某类时间触发进程执行的选择，则仓库是一个传统型数据库；若中央数据结构的当前状态触发进程执行的选择，则仓库是一个黑板系统。图 4.5 是数据库和黑板系统的组成。

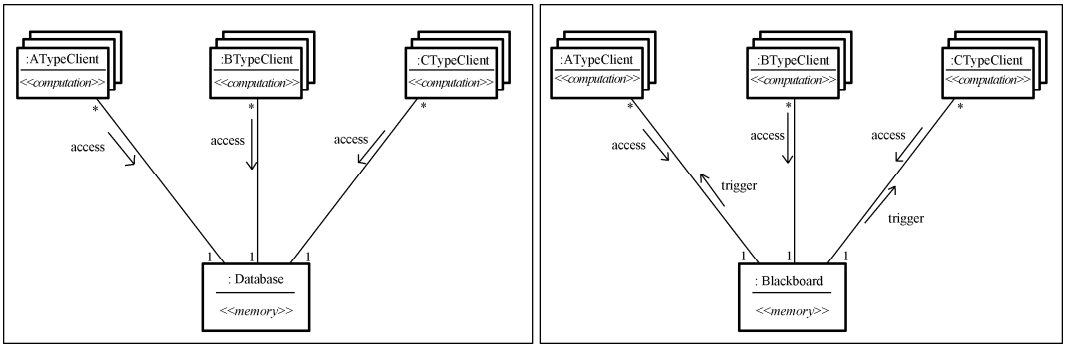


图 4.5　数据库和黑板系统的组成

图 4.5 中右图为黑板系统，其主要由三部分组成。

(1) 知识源。知识源中包含独立的、与应用程序相关的知识，知识源之间不直接进行通信，它们之间的交互只通过黑板来完成。

(2) 黑板数据结构。黑板数据是按照与应用程序相关的层次来组织的、解决问题的数据，知识源通过不断地改变黑板数据来解决问题。

(3)控制。控制完全由黑板的状态驱动，黑板状态的改变决定特定的知识的改变。

黑板系统的传统应用是信号处理领域，如语音和模式识别，另外的应用包括松耦合代理数据共享存取等。

4.3.4 正交软件结构

正交软件结构由组织层和线索的构件构成，其中，层是由一组具有相同抽象级别的构件组成；线索是子系统的特例，它由完成不同层次功能的构件组成，通过相互调用来关联，每一条线索完成整个系统中相对独立的一部分功能。每一条线索的实现与其他线索的实现无关或关联很少，在同一层中的构件之间不存在相互调用。

如果线索是相互独立的，即不同线索中的构件之间没有相互调用，那么这个结构就是完全正交的。因此，从定义可知，正交软件结构是一种以垂直线索构件为基础的层次化结构，其基本思想是把应用系统的结构，按功能的正交相关性垂直分割为若干条线索(子系统)，线索又分为几个层次，每条线索由多个具有不同层次功能和不同抽象级别的构件构成，各条线索的相同层次的构件具有相同的抽象级别，线索之间是相互独立正交的。正交软件结构的框架如图 4.6 所示。

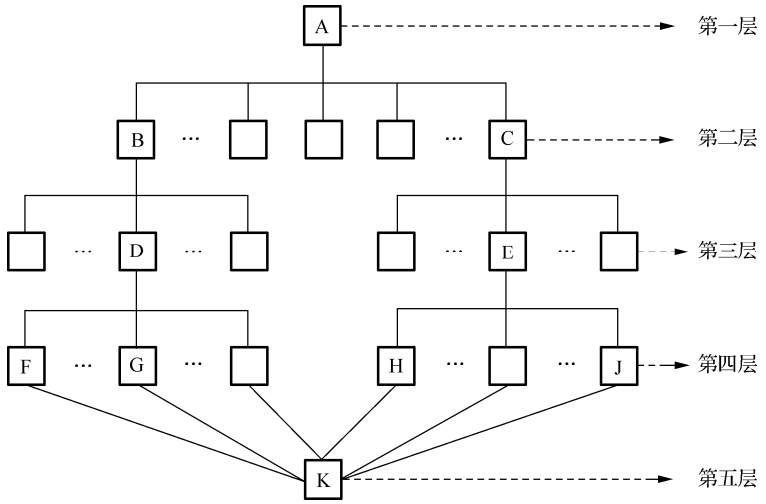


图 4.6　正交软件结构的框架

图 4.6 所示是一个三级线索、五层结构的正交软件结构框架图。图中，ABDFK 组成了一条线索，ACEJK 也是一条线索。B、C 处于同一层次中，不允许进行互相调用；H、J 处于同一层次中，也不允许互相调用。第五层可能是一个物理数据库连接构件或设备构件，可供整个系统公用。

在软件演化过程中，系统需求会不断发生变化。在正交软件结构中，因为线索的正交性，每个需求变化仅影响某一条线索，而不会涉及其他线索。这样，就把软件需求的变动局部化了，产生的影响也被限制在一定范围内，因此容易实现。正交软件结构是一种典型的调用/返回体系结构。

4.3.5 客户机/服务器结构

客户机/服务器(Client/Server，C/S)结构在信息产业中占有重要的地位。网络计算经历了

从基于宿主机的计算模型到客户机/服务器计算模型的演变，在集中式计算技术时代广泛使用的是大型机/小型机计算模型，它是通过一台物理上与宿主机相连的非智能终端来实现宿主机上的应用程序。

C/S 软件结构是基于资源不对等，且为实现共享而提出来的技术，C/S 结构定义了客户机如何与服务器相连，以实现将数据和应用分布到多个客户机上。C/S 结构由三个主要部分组成：服务器、客户应用程序和网络，如图 4.7 所示。

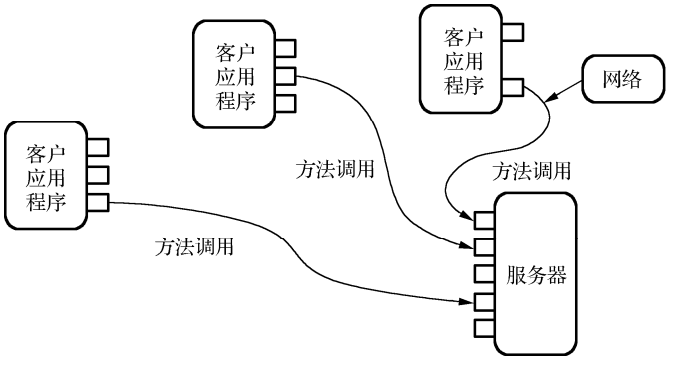


图 4.7　C/S 结构示意图

服务器负责有效地管理系统的资源，其任务集中于服务器访问与并发性控制、服务器安全性、服务器的备份与恢复和全局数据完整性规则。客户应用程序的主要任务是提供用户与服务器交互的界面、向服务器提交用户请求并接收来自服务器的信息、利用客户应用程序对存在于客户端的数据执行应用逻辑要求。网络的主要作用是完成服务器和客户应用程序之间的数据传输。

C/S 结构将应用一分为二，服务器（后台）负责数据管理，客户应用程序（前台）完成与用户的交互任务。服务器为多个客户应用程序管理数据，而客户应用程序发送、请求和分析从服务器接收的数据。C/S 结构主要便于分配系统的客户应用程序和服务器构件分别运行在不同的计算机上，系统中每台服务器都可以适合各构件的要求，这对于硬件和软件的变化显示出极大的适应性和灵活性，而且易于对系统进行扩充和缩小。在 C/S 结构中，系统中的功能构件充分隔离，客户应用程序的开发集中于数据的显示和分析，而服务器的开发集中于数据的管理，不必在每个新的应用程序中都要对数据存储进行编码。

4.3.6　浏览器/服务器结构

浏览器/服务器（Browser/Server，B/S）结构主要是利用不断成熟的 WWW 浏览器技术，结合浏览器的多种脚本语言，用通用浏览器实现原来需要复杂的专用软件才能实现的强大功能，并节约开发成本。在 B/S 结构中，除了数据库服务器，应用程序以网页形式存放于 Web 服务器上，用户运行某个应用程序时只需在客户端上的浏览器中输入相应的网址，调用 Web 服务器上的应用程序并对数据库进行操作，完成相应的数据处理工作，最后将结果通过浏览器显示给用户。可以说，在 B/S 结构的计算机应用系统中，应用程序在一定程度上具有集中特征。

基于 B/S 结构的软件，系统安装、修改和维护全在服务器端解决。用户在使用系统时，仅

仅需要一个浏览器就可运行全部模块，真正实现“零客户端”功能，也很容易在运行时自动升级。B/S 结构还提供了异种机、异种网、异种应用服务的联机、联网、统一服务的最现实的开放性基础，其结构如图 4.8 所示。

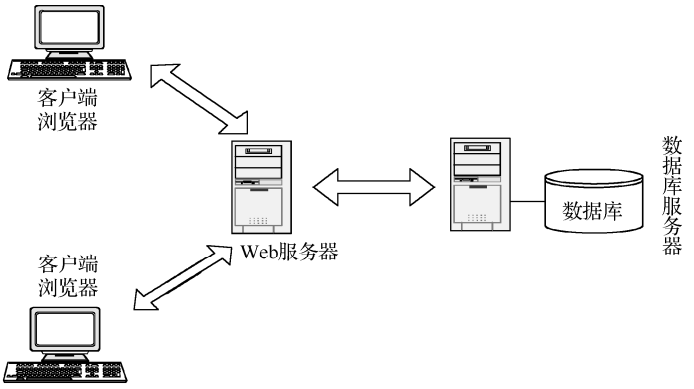


图 4.8　B/S 结构

客户端上只有浏览器负责解析和显示交互数据，但是不包括有关业务本身的处理逻辑。Web 服务器层包含具体的业务处理逻辑，而处理所需的数据则要从表示层或数据层取得。数据层就是数据库管理系统，负责管理对数据的读写。数据库管理系统必须能迅速执行大量数据的更新和检索。现在的主流是关系型数据库管理系统(RDBMS)。因此，一般从功能层传送到数据层大多使用 SQL 语言。

与 C/S 结构相比，B/S 结构增加了一个应用服务器，可以将整个应用逻辑保存在应用服务器上，客户端的压力大大减轻，负荷被均衡地分配给了服务器。由于这种结构不再需要专用的客户端软件，因此也使技术维护人员从繁重的安装、配置和升级等维护工作中解脱了出来，可以把主要精力放在服务器程序的更新工作上。同时，使用 Web 浏览器作为客户端软件，界面友好，新开发的系统也不需要用户每次都从头学习。而且，这种三层模式，层与层之间相互独立，任何一层的改变都不影响其他层原有的功能，所以可用不同厂家的产品组成性能更佳的系统。

4.3.7　MVC 结构

MVC 全名是 Model View Controller，是“模型(Model)－视图(View)－控制器(Controller)”的缩写。MVC 是一种软件设计典范，用业务逻辑、数据、界面显式分离的方法组织代码，将业务逻辑聚集到一个部件里面，在改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑。

MVC 最开始存在于桌面程序中，其结构如图 4.9 所示。M 是指模型，V 是指视图，C 是指控制器，使用 MVC 的目的是将 M 和 V 的代码分离，从而使同一个程序可以使用不同的表现形式。例如，一批统计数据可以分别用柱状图、饼状图来表示。C 存在的目的则是确保 M 和 V 的同步，一旦 M 改变，V 应该同步更新。

视图是用户看到并与之交互的界面。对传统的 Web 应用程序来说，视图就是由 HTML 元素组成的界面，在动态的 Web 应用程序中，HTML 依旧在视图中扮演着重要的角色，但一些

新的技术已层出不穷，包括 Adobe Flash 和像 XHTML、XML/XSL、WML 等一些标识语言和 Web Services，MVC 能为应用程序处理很多不同的视图。

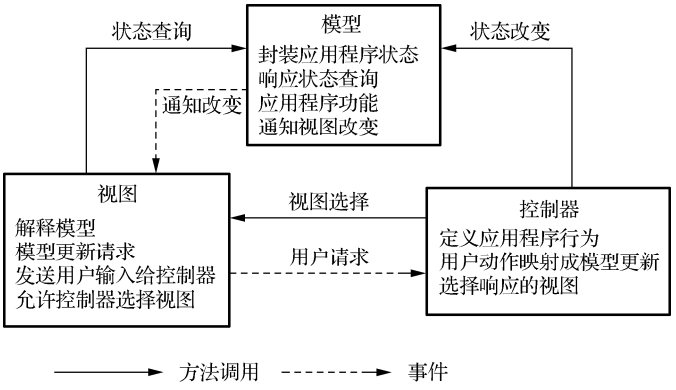


图 4.9　MVC 结构

模型表示企业数据和业务规则。在 MVC 中，模型拥有最多的处理任务。例如，它可能用像 EJBs(Enterprise JavaBeans)这样的构件对象来处理数据库，被模型返回的数据是中立的，模型与数据格式无关，这样一个模型能为多个视图提供数据，由于应用于模型的代码只需要写一次就可以被多个视图重用，因此减少了代码的重复性。

控制器接收用户的输入并调用模型和视图完成用户的需求，所以当单击 Web 页面中的超链接和发送 HTML 表单时，控制器本身不输出任何结果，也做任何处理。它只是接收请求并决定调用哪个模型去处理请求，然后再确定调用哪个视图显示返回的数据。

4.4　软件架构设计

在软件架构设计的初期，要考虑软件所处的环境，即应该定义与软件交互的外部实体(其他系统、设备、人等)和交互的特性。这些信息一般是在建立分析模型的阶段获得的，而所有其他的信息都是在需求工程阶段获得的，第 3 章中的图 3.18 可以辅助我们建立一个粗略的模型框架。一旦建立了软件的环境模型，并且描述出所有的外部软件接口，开发者就可以通过定义和求精，实现架构的构件，描述系统的结构。这个过程不停地迭代，直到获得一个完善的架构为止。

通过软件架构，系统将逻辑关系密切的单元划分到一起，形成系统的逻辑划分，有利于后续独立的开发和管理。这个划分经常是基于类模型进行的，并可参照一些设计优化方法形成更合理的组织方式，达到模块内部的高内聚和模块间的低耦合。软件架构对应的实现就是将软件使用“包(Package)”进行构造，每个包对应某种专属的功能，并尽可能独立。包中的类互相紧密配合，完成包的功能，每个包与其他包中含有的类之间的接口应该尽可能简单，降低它们的耦合性。

4.4.1　包及其结构

包与包之间可以相互嵌套构成包的层次关系，在 Java 的类库中，就存在很多这样的情况，图 4.10 就是其中的一个例子：在图形界面包 swing 中嵌入了一个用于文本显示或处理的 text

包。除了一般的文本处理类，该包又包含了专门针对 html 和 rtf 格式处理的两个包，在 html 包中又含有一些专门用于 html 的解析类，这些解析类又进一步被组织到其中的 parser 包中。这里要注意，嵌套的包有两种绘制方式：直接嵌套或者通过一个带圈的十字连接符嵌套。

如图 4.10 所示，包在 UML 中都以标签页的形式显示，若只需要显示包的名字，则可以直接将包名写在每个包的中间位置；若还需要显示包中含有的内容(如所含的类)，则将需要进一步显示的图形或元素放置在包的中间矩形部分，包的名字则显示在标签页的头部。包全名的构成包含所有的前缀内容，形如：

```
MainPackageName.SubPackageName.SubPackageName
```

理论上，这种形式包含的包结构可以任意长地进行组织，通过包的全名能够较清楚地了解包所处的位置，如 swing 包中的 text 包中的 rtf 包。

通过包，可以对类进行合理的划分，图 4.11 显示了系统中业务相关的类通过界面、业务和数据构成了一个层次架构。界面层中包含的类多为图形显示或与用户交互相关的类；业务层中包含所有与业务逻辑相关的类，在分析模型的构建时，主要针对的就是此层中的类及其关系；数据层的类主要负责数据存储，即负责为对象的持久化服务，包括存和取两个方向的操作，这层又称数据访问层，其中的对象又称 DAO(Data Access Object)。

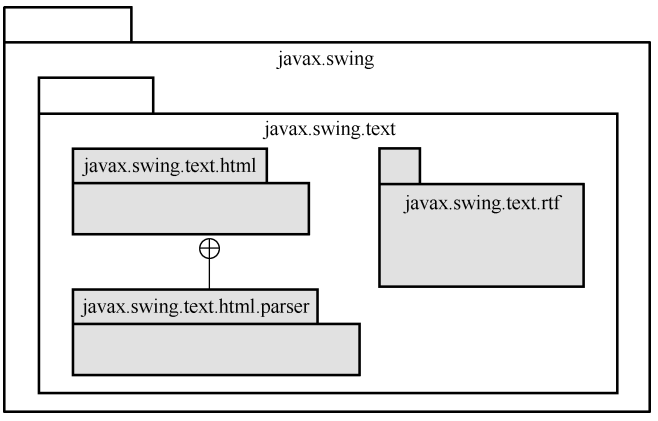


图 4.10　Java 中的包层次示例

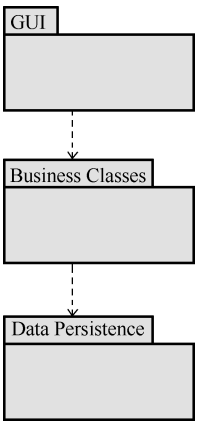


图 4.11　三层架构的包结构

在图 4.11 中，包之间的虚线箭头描述的是层间的使用(依赖)关系，若有包 A 指向包 B 的虚线箭头，则表示 A 中的类通过某种方式使用了 B 中的类。包结构中一个重要的要求是在包之间不能出现循环的依赖，即不能从包 A 出发，通过若干个依赖关系又回到包 A。这种循环依赖在类图中也同样需要避免，否则会导致逻辑复杂、实现困难。例如，如果包 A 使用包 B，那么对包 B 中某个类的修改意味着包 A 中的某些类可能会受到影响，反之则不会。但是，在循环依赖的情况下，对于任何包中的修改，都要对各个依赖方向上的包进行循环检查，这显然是非常烦琐和不可取的。

4.4.2　包结构设计

作为三层架构的拓展，可能也会存在多层架构。如有时可能需要建立一个工具包来对所有其他的包提供基础性的服务支持。在类似这样的包中，可以存储一些全局类，但应尽量减少这些全局类的数量。在全局类中，通常还会涉及一些与具体项目相关的业务类型，如其他业务

类需要的枚举类型等。还可以放置一些主要的项目配置信息，如文件路径配置等，这些信息同样可以在全局类中存储，并在多个包中使用。在这些全局类中，我们习惯使用静态变量和静态方法实现对它们的组织和管理。

图 4.12 描述了初始的项目管理系统的层次架构。其中，Projects 包和 ProjectMembers 包是两个业务层的包，DAO 包是数据层的包。关于数据存储的实现，会在 9.5 节中的数据持久化部分进行介绍。界面层的 GUI 包中设置了一个控制类 GUIControl，其作用是将用户事件的处理向不同的业务接口类分发，同时也使得该架构有了 MVC 的雏形。GUI 包中还有一些 Mask 类，它们基本是按照对应的业务类设置的，其作用是提供显示和控制需要的业务信息和功能，如交互数据的校验和规则等，对它们的理解可参考 5.4 节。包的结构需要满足基本的非循环依赖要求，在后续的结构调整和优化中，也要注意满足这个要求。

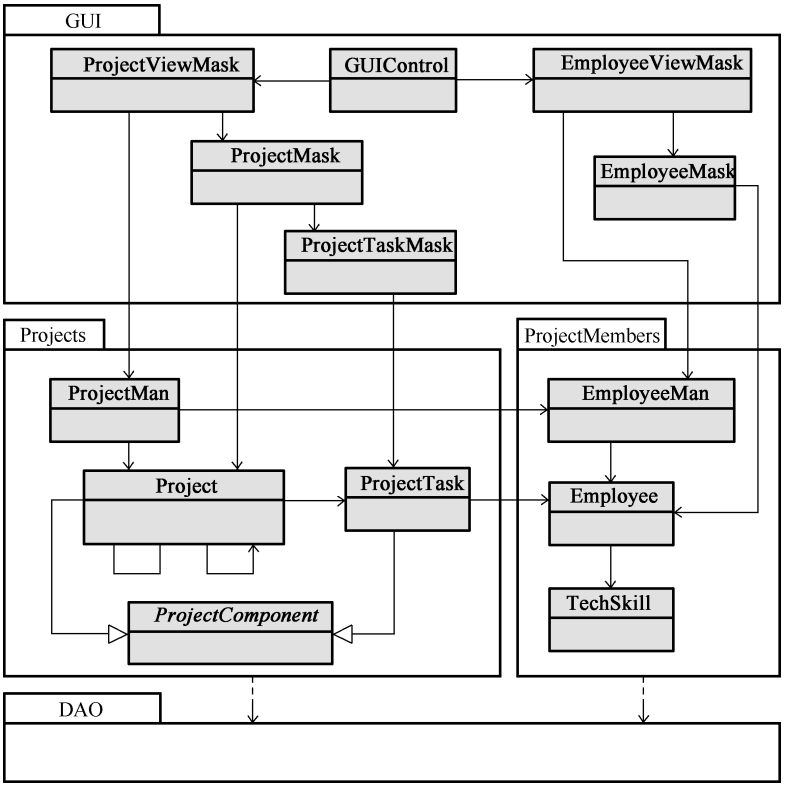


图 4.12　项目管理系统的初始包图

图 4.12 中虽然没有循环依赖，但存在的主要问题是很多位于不同包的类之间的依赖关系较多，使得包与包之间的关系变得复杂。进一步优化的目标就是要将类更加合理地在包之间划分和组织，以提高各部分的独立性，从而达到更高的内聚性。基于这样的考虑，对初始包进行设计的调整，如图 4.13 所示，将 GUIControl 类进一步扩展，加入控制协调类，其职责主要是控制业务包中类的访问，使得 Mask 类不再直接访问业务类。这种设计思想是设计模式的门面模式的体现。另外，图 4.13 中还对所有业务类的 Mask 类提炼出一个 Mask 抽象类 Abstract Mask，提供所有 Mask 类的统一形式。在 Projects 包和 ProjectMembers 包中分别加入了两个接口，目的是进一步降低与 GUI 包之间的耦合性。

最后，对包的宏观结构进行调整，突出整体的层次，如图 4.14 所示。业务包中的 ProjectMan 和 EmployeeMan 中主要包含一些负责控制和管理的类，而 ProjectData 和 EmployeeData 中包含业务实体类。

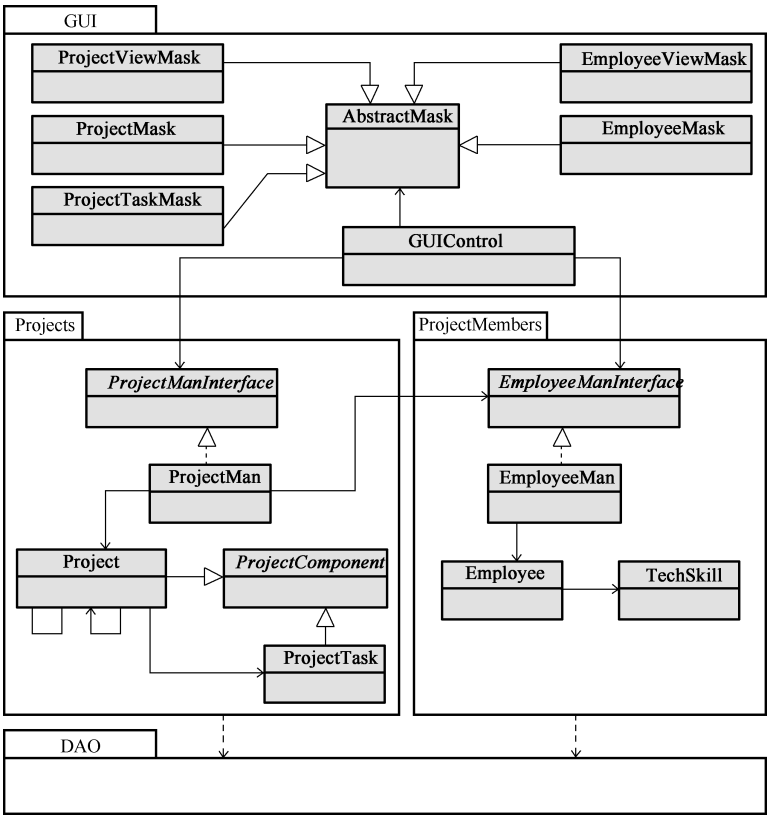


图 4.13　包的调整

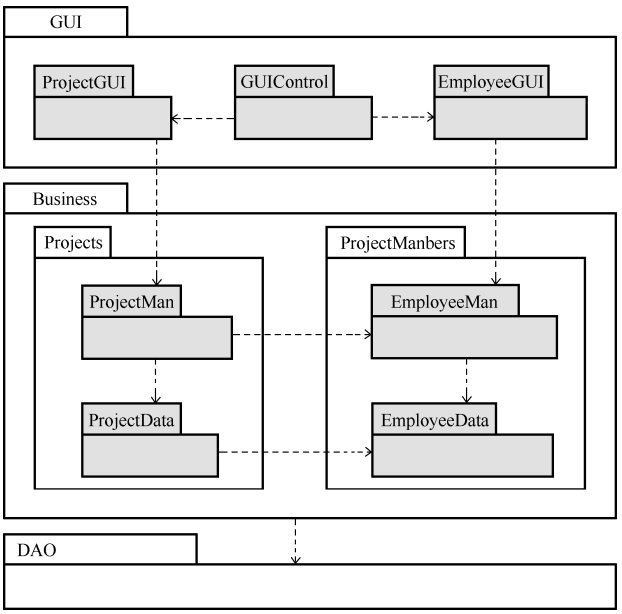


图 4.14　包的宏观结构调整

4.5 习题

某报告生成系统的处理流程如下：首先从配置文件中加载必要的参数，然后读取 Word 模板文件，模板文件定义了报告的基本格式及相关数据变量的定义及其引用。模板中的内容以数据流的方式依次经过预处理、数据加载及文档生成三个环节，最后产生 doc 或 pdf 格式的报告，如图 4.15 所示。该系统的架构设计采用何种架构模式较为合适？为什么？这种模式有什么优点(两个)？

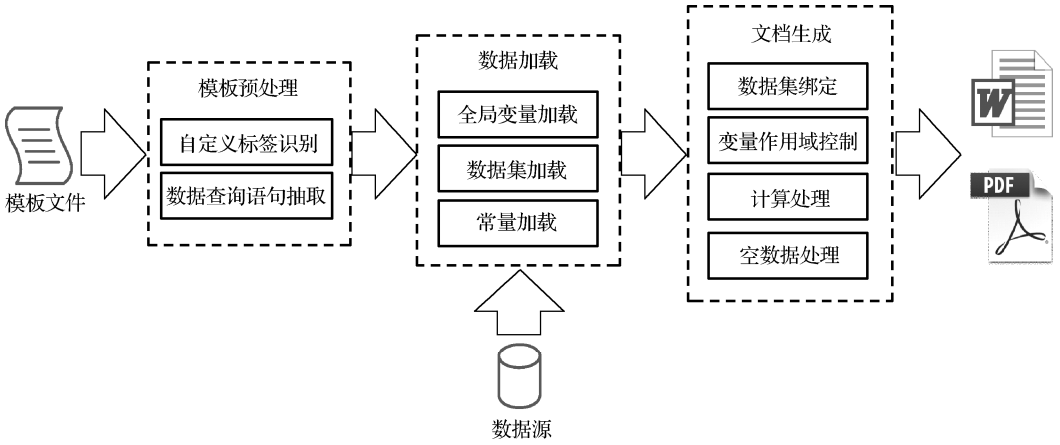


图 4.15　某报告生成系统的处理流程

第 5 章　类的分析与设计

类的分析与设计的目的是将需求阶段得到的信息加以整理和归纳，以开发者的视角抽象出相对稳定的静态结构，并逐渐转换为开发者的语言。也就是要对未来系统的功能进行总体上的概述，并使用 UML 类图进行表达。UML 支持的面向对象的设计方法允许在开始阶段粗略地对模型进行构建，后续再通过迭代，逐级具体化，是一个逐步求精的设计过程。初期创建的分析类图的目的就是覆盖所有需求，并通过优化，尽量保持其业务结构的稳定性，然后通过修订和丰富细节逐渐过渡到详细设计，并最终转换为成功的物理实现。

本章首先从文本描述的需求出发，详细介绍使用系统的方法生成UML表示的类图的过程，并通过迭代的方式逐步细化，结合分析人员的经验，将需求要点向类图传递并使之丰富。

5.1　基本类的确定

假定读者已经熟悉了面向对象软件开发的基本概念和原理，如类、方法、继承、多态及抽象类等，这些概念在这里只做简单介绍，建议不太熟悉相关概念的读者参考一些面向对象的书籍。

面向对象的分析和设计过程主要是将需求阶段捕获到的结果转换为模型的表示，并基于这些模型进行设计的优化，作为后续代码的实现基础，需求规格说明书是进行面向对象建模的前提条件。

在软件系统运行时，类将被实例化成对象(Object)，对象对应于某个具体的事物，是类的实例(Instance)。面向对象的分析首先要基于需求规格说明书，寻找系统中参与业务处理的对象和类。这些业务处理过程一般都围绕业务对象展开，下面介绍的内容可用于识别主要的业务对象。

类图(Class Diagram)使用出现在系统中不同的类来描述系统的静态结构，包括业务类及其关系。在类的分析与设计阶段，类通常可以分为实体类(Entity Class)、控制类(Control Class)和边界类(Boundary Class)三种。

1. 实体类

实体类对应系统需求中的每个实体，它们通常需要在存储介质上持久保存，如使用数据库表或文件来记录。实体类既包括存储和传递数据的类，又包括操作数据的类。实体类来源于需求分析中的名词，如项目、任务等。

2. 控制类

控制类用于体现应用程序的执行逻辑，提供相应的业务操作，将控制类抽象出来可以降低交互层和存储层之间的耦合度。控制类一般是由动宾结构的短语(动词+名词)转化来的名词，如增加商品可以对应一个“商品增加”类，用户注册可以对应一个“用户注册”类等。

3. 边界类

边界类用于对外部用户与系统之间的交互对象进行抽象，主要包括界面类(如对话框、窗口、菜单等)及与外部系统的数据交换类(如同步、缓存等)。

在类的分析和设计的初级阶段，通常首先识别出实体类，绘制初始类图，此时的类图也可称为领域模型，主要含有实体类及它们之间的相互关系。

系统中的对象在表示上应具有唯一的标识 OID 及若干相关的属性(Attribute)。例如，一个具体的“项目”的名字为“考勤系统”，项目经理为“王楠”，项目开始日期为“2018 年 10 月 20 日”，这些属性又称实例变量(Instance Variable)。同类对象的共同结构可通过它们的类进行说明，所有的实例变量都可包含在类中，作为类的初始信息。因此，对于“项目”类来说，它是所有具体“项目对象”的一个模板。

5.1.1 类的识别

类的寻找和细化是一个迭代的过程，开始时总是建立类的雏形，带有基本的实例变量，然后不断补充新的类和信息并逐渐扩展，最后发展为更多的类和实例变量。

需求规格说明书是寻找实体类的直接来源，通过文字分析确定是否存在某个对象或者属性的线索，这是一种快速而实用的分析方法，其实质是按照语法分析将名词标注为对象的候选，将形容词标注为属性(实例变量)的候选。除了需求的功能描述，数据字典(业务术语词汇表)还是类信息的重要来源，这些与业务术语相关的类通常就是实体类，代表的都是在系统中需要管理的实际的业务对象。下面针对“软件项目管理系统”的需求描述进行实体类的分析和识别。

(1)针对需求 R1.1 项目创建，首次迭代主要关注的是类及其属性。通过 R1.1 和词汇描述，可以首先识别出项目类，其含有项目编号、项目名称、项目起止时间及预计工作量等属性。

从具体需求的文本源中得出的类、属性或者方法需要在文档中进行记录，这是实现需求跟踪的需要，应尽可能通过 CASE(Computer Aided Software Engineering，计算机辅助软件工程)工具对需求源和类图元素之间的对应关系进行管理。图 5.1 给出的就是利用 CASE 工具在文字性的需求陈述和抽象出来的业务元素之间进行映射和管理的一个示例。

(2)在需求 R1.2 数据存储和 R1.3 中，虽然提到了用户期望的功能性需求，但却不能直接导出相关的业务类和对象。R1.3 项目选择的需求需要说明一下，因为这个需求需要对涉及的项目进行统一管理，因此需要一个管理类，在二次迭代中可以考虑加入。

(3)针对需求 R1.4 子项目创建，对 R1.4 子项目创建分析后可知，子项目为项目的一个实例变量，并根据数据字典中对子项目的描述“子项目与项目结构相同”，可以暂时将子项目识别为一个备选的类，其属性与项目类的属性相同。

(4)针对需求 R1.5 子项目与项目，分析可知，子项目与项目是同义词，所以，子项目不需要单独设置一个类。对于同义词，有“异形同义”的情况，还有“同形异义”的情况，这应在需求分析阶段的数据字典中进行标识。在建模时，需要再次考虑这些情况，明确它们是否应以相同的处理方式在设计中进行展现。

(5)针对需求 R1.6 项目信息编辑，可识别出项目类的属性还应包括实际开始时间、最新计算出的结束时间、预计工作量、项目备注等。

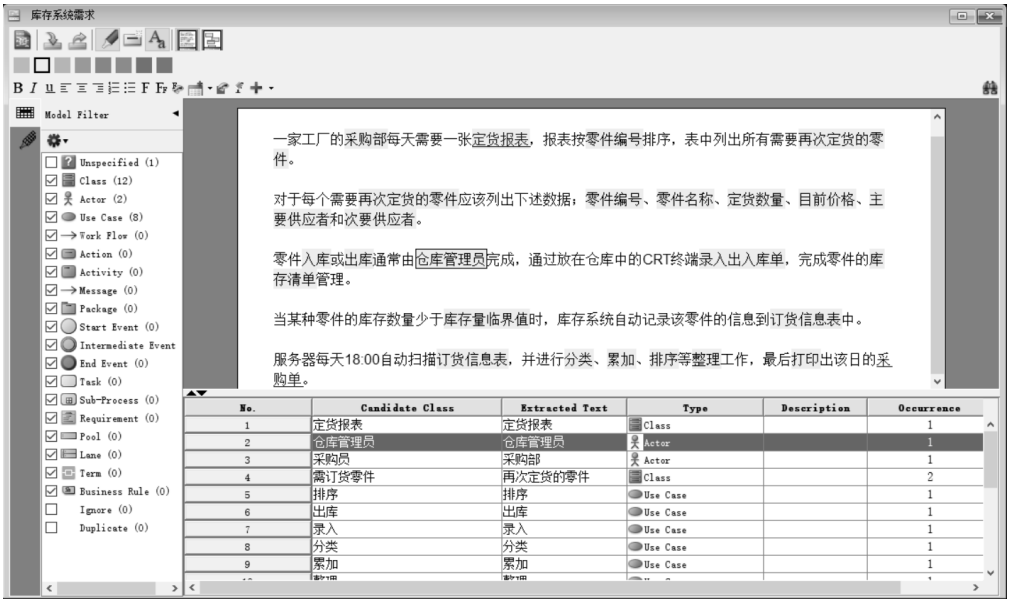


图 5.1　从需求陈述中提取业务元素

(6)针对需求 R1.7 项目任务添加，可识别出来的信息包括项目类的“任务”属性、项目任务类及其属性，包括名称、责任人、计划与实际工作量、计划和实际的开始时间、计划和实际的结束时间、完成进度。

同样，项目类也应具有完成进度及实际工作量等属性，并且它们的值能够通过其他相关信息计算出来，这样的属性称为依赖属性。另外，项目属性“任务”与其他只能取单一值的属性不同，它的取值可以是一个集合，并且可以为空。

(7)针对需求 R1.8 项目任务选择和 R1.9 项目任务编辑，没有新的信息被识别出来。

(8)针对需求 R1.10 对其他项目的依赖，可识别出项目类应具有属性“前驱项目”。

5.1.2　初始类图

将到目前为止发现的所有信息融入一个类图中加以表示，形成初始类图，如图 5.2 所示。每个类使用一个矩形框表示，其上面的方框内是类名，与中间的实例变量部分有一线之隔，实例变量的描述格式为“变量名:类型”，命名的习惯通常是将类名首字母大写，属性名的首字母小写。属性名前面的短横线表示其可见性为私有的(private)。这是根据类的一个基本原则——封装性设置的，即其他类对私有属性的读取只能通过该类提供的方法进行，不能直接进行操作。类内部的方法对类内的私有属性则可以直接操作，而不受任何访问权的限制。其他可见性还包括公有的(public)，用“+”作为前缀表示，该属性对所有类可见；受保护的(protected)，用“#”作为前缀表示，对该类的子类可见；包的(package)，用“~”作为前缀表示，只对同一包中声明的其他类可见。

图 5.2 中 Project(项目)类的两个实例变量“compeletePct(完成进度)”和“effortReal(实际工作量)”前都有一个斜线“/”，这表示它们是依赖属性，其取值是通过其他相关变量的值计算而来。在后续的开发中，可能需要进一步确定它们是否需要真的进行实现，这里有两种选择：由开发者维护数据的一致性，或者在需要时实时计算。

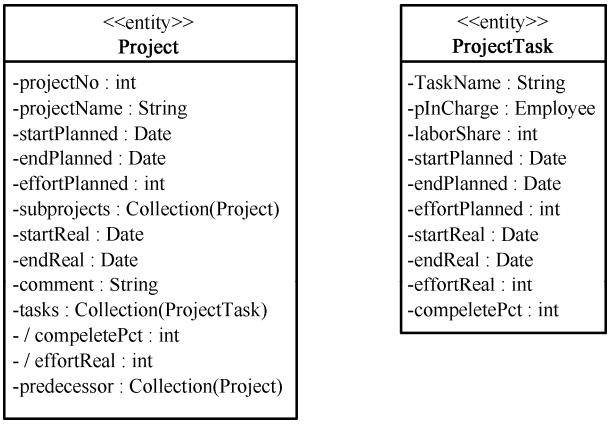


图 5.2　初始类图

类中实例变量的类型可以按照 UML 规范中预定义的类型选择，如果编程语言已经确定，也可以直接使用该语言支持的类型。在分析类图中，通常先省略变量的类型部分，暂时忽略这些实现细节。UML 中提供了针对实体类的构造型<<entity>>，也可以使用实体类的便捷表示，如图 5.3 所示，两者是等价的。

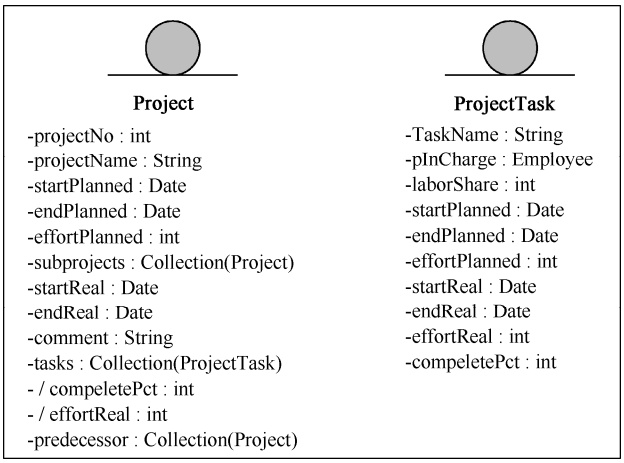


图 5.3　实体类的便捷表示

5.1.3　类的方法

除了实例变量的说明，类中还包含方法，又称操作或对象功能，它们为业务计算或实例变量值的读写提供了服务。因为一个对象中所有实例变量值的组合构成了该类的状态集合，所以通过这些方法可以对类的状态进行修改。

方法的典型作用是提供对实例变量值的访问和修改，因为它们不涉及具体业务，所以在分析模型中通常不会考虑，而是在后续的实现阶段再确定它们存在的必要性。除此之外，类中可能还会提供一些业务方法，它们需要其他的信息(内部或外部)来对某些业务数据进行计算，完成响应的业务功能。例如，Project 类中的方法 computeAllocatedEffort()，用来计算一个项目中已经对其包含的任务和子项目分配的工作量值。由于该方法只需要 Project 类的内部信息

即可，所以将其放置于 Project 类中是合适的，否则将会破坏该类的独立性。

在初始的迭代中，首要的任务是识别出与业务紧密相关的实体类，实体类一般不存在与其他类的交互，是系统处理业务的基础信息和载体，因此彼此之间也较为独立。在后续的迭代中，在对业务进行更加深入的分析后，可能会识别出更多的关于实体类间的交互和作用方式，当业务较为复杂时，有效地提取和安置这些类的方法可能会有困难，这时可以借助 UML 其他动态的模型辅助分析。

5.1.4 类的关系

下面考虑类与类之间的关系如何表达。类与类之间最常见的关系是静态的关联关系（Association），表达了它们的对象之间的一种“持有”联系。例如，一个项目对象与多个任务对象的包含关系，使用这种关联关系进行表达就比较合适，其表现方式为类间的一条直线。关联关系的表达可以是双向的，实现时，一般通过类的实例变量来存储持有的对象。另外，类与类之间还存在一种动态的依赖关系，体现对象的瞬时使用关系，即一个类的实现需要另一个类的协助，但并不需要进行保持，因此，尽量避免双向的互相依赖。依赖关系的描述形式是带箭头的虚线，箭头指向被使用者。实现时，方法的局部变量、方法的参数或者对静态方法的调用等瞬时表现都是依赖关系的情况。

由于在类图中，我们一般只考虑类的关联关系，因此后面的内容将主要围绕关联关系进行介绍。关联关系本身也可以含有一些附加的额外信息，如图 5.4 中所示，它们也需要逐渐地完善。

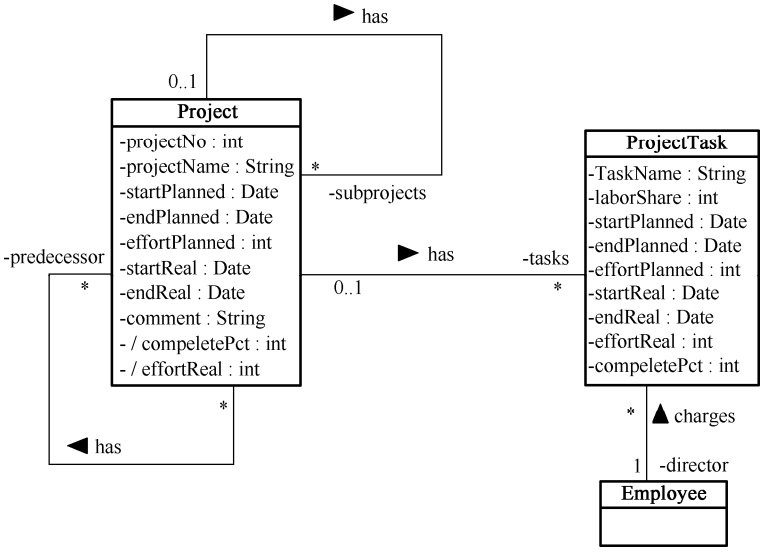


图 5.4　加入关联关系的类图

Project 类的属性“任务”其实无须显式地给出，因为它可以间接地通过与 ProjectTask（项目任务）类之间的关联关系“has”进行体现，其中的黑色三角表示关系的方向。该关系在靠近任务类的端点处标注了“-tasks”的角色，其中，“-”表示私有，另一端则没有标注角色，这意味着关联关系的角色是可选的，默认为类的名字。另外，关联关系具有“多重性”，一个星

号(*)表示任意多的意思，即可以为空值、一个或多个项目任务。在关联的另一个方向，该关联表示每个任务对象只属于一个项目或者没有对应的项目。关联关系中的基数(多重性)的几种典型的表达方式可总结如下，其中数字 3 和 7 表示泛指，可以为其他数字。

(1)“*”：任意多个(包括 0 个)对象。

(2)“1”：只有 1 个对象。

(3)“3”：正好 3 个对象。

(4)“1..*”：最少 1 个，也可能为多个对象。

(5)“3..*”：至少 3 个，也可能为多个对象。

(6)“0..1”：0 或 1 个对象。

(7)“3..7”：3 到 7 个对象。

在图 5.4 中，我们还看到一个类可以具有指向自己本身的关联关系，称为自反关联(Reflexive Association)。例如，Project 类具有两个自反关联，其中一个表示：一个 Project 对象可以具有多个前驱对象，同样一个 Project 对象可以作为多个 Project 对象的前驱；另一个自反关联请读者自行思考解释。

在 ProjectTask 类中含有一个具有 Employee(员工)类型的实例变量，用以保存对应的负责人信息，这里也是通过关联关系进行表述的，因为 Employee 类型本身就是一个类，而不是一个简单类型，如布尔型、整型、浮点型或字符串类型。当然，在类模型的构建过程中，开发者需要斟酌，对于员工来说，是需要为其单独建立一个独立的、具有属性的实体类，还是只需要使用一个 String 类型表示其名字即可。一般来说，在类模型中，若只使用到一些实际编程语言中的标准类型或类库中的标准类作为变量类型，则将它们直接视为简单变量类型即可，不必设置单独的类，例如，类图中的日期类型。

类图的创建和调整是一个高度动态的过程，在这个过程中经常会有新的类和关联关系被识别出来，并反复修改，甚至丢弃。一种实用的方法是将每个类描绘在卡片上并挂在一面可擦写的磁性白板上，从而可以很容易地将其拿下或贴上。类之间的关联关系也能较方便地绘制和擦除，而且可以将不同版本的类图拍照，并以照片的形式进行保存。这种在卡片上绘制类的方法又称 CRC(Class Responsibility Collaborators)，卡片上主要记录类的名字、包含的属性、主要职责、有消息交互的其他类等，如图 5.5 所示。



图 5.5　CRC 示例

一般来说，根据项目的复杂程度和建模人员的经验，在充分和深入地了解业务的情况下，通过多次迭代才能将类图和关联关系等信息完全、合理地描述出来。

5.1.5 类与对象

前面已经对类与对象之间的关系进行了简单介绍，类代表一种结构，对象是类实例化后的结果，具有类结构中所有的成员变量，并且对它们赋予了具体的值。对象也可以使用 UML 对象图进行描述，图 5.6 给出了一个 Project 类并实例化了它的一个对象，图中只是为了对比，在实际的开发中，类与其对象同时出现的情况并不常见。与关联关系类似，对象间的关系在对象图中一般使用实线进行连接，它们其实也是类的关联关系的实例化结果。

对象在 UML 中通常使用“对象名:类名”的形式进行指代，注意其具有下画线。当在上下文语境清晰的情况下，对象名可以省略，如“:Project”表示一个匿名 Project 类对象。

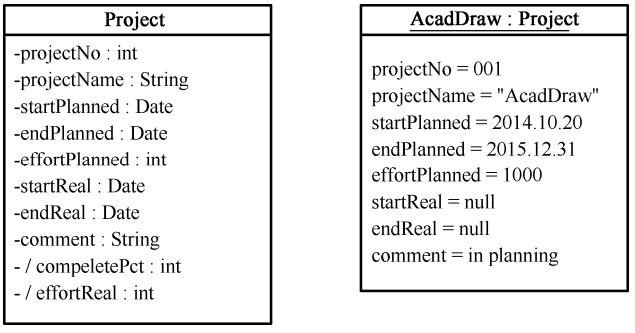


图 5.6 类与对象

5.2 类的细化

在接下来的迭代中，将重新审视并分析需求陈述和词汇表中提到的功能与对象之间的对应关系。深入业务层次，着重理清业务的管理和控制流程，并将这些信息落实到具体的分析类中。

5.2.1 管理类和控制类

首先对管理类和控制类做一个区分，尽管有些文献中认为两者具有相同的含义，但为了便于分析和设计，我们将管理类理解为对同类对象的协调和管理，主要负责同类对象的创建、代理访问其他对象的信息等。管理类必须能够提供对其所管辖的所有对象统一的处理方式，一般是借助某种容器实现。实体类也可以实现管理类的功能，如 Project 类就应该提供对所属所有任务的管理。

控制类用于对一个或几个用例所特有的控制行为进行建模。控制类通常控制和协调不同对象的行为，用来封装用例的特有行为。复杂的用例一般都需要一个或多个控制类来协调系统中对象的行为，借助控制类可以有效地将边界对象与实体对象分开，让系统更能适应其边界内发生的变更。控制类还将用例所特有的行为与实体对象分开，使实体对象在用例和系统中具有

更高的复用性。控制类的识别一般采用的方法是：先对所有的用例进行分析，每个用例对应产生一个控制类，用来对该场景中需要的对象进行管理和协调。在对控制类进行设置和细化的过程中，有以下两点建议。

(1)每次只考虑一个任务，只向控制类添加与该任务相关的方法和方法需要的实例变量。

(2)类与类之间尽可能保持较少的联系，这样可以降低接口的数量。

下面重新梳理需求，进一步识别和补充必要的类和类的方法。这里重点通过对句子中的动词和更多对象的寻找来辅助分析可能的类和方法。

(1)R1.1 项目创建：此需求在执行时需要对应创建一个新的项目对象。创建对象时需要以参数的形式传递若干数据给新类。项目对象是通过构造函数进行创建的，这里只给出那些与标准构造函数不同的函数，即具有参数的构造函数。将加入以下需要 4 个参数的项目类的构造函数：

```
Project(String, Date, Date, int)
```

(2)R1.2 数据存储：这里在控制上有一个隐含的需求，作为功能应该为相关的实体类提供必要的 set 函数，其作用是最终向对象写入录入的值。

(3)R1.3 项目选择：这里需要一个新的管理类 ProjectMan(项目管理)类，其作用是管理所有项目对象及当前被选择的项目。所以，该管理类应具有所有项目的列表和被选择的项目的两个直接的属性，其包含的方法可以加入对属性“当前被选择的项目”的 set 方法。

(4)R1.4 子项目创建：Project 类需要对包含的子项目进行管理，因此 Project 类既是一个实体类，又是一个管理类，Project 类是子项目的一个容器，其添加(add)和删除(remove)操作是常用的方法。对于本需求，需要在 Project 类中加入一个函数 addSubproject(Project): void，并约定若其返回值为空值，则表示创建子项目的过程没有成功完成。

(5)R1.7 项目任务添加：项目应具有 addTask(ProjectTask): void 方法，进一步为 ProjectTask 类加入构造函数：ProjectTask(String, Date, Date, int, int)。这里隐含存在一个功能，即对项目实际工作量的计算，还有对完成进度的计算。因此，在类中加入方法：realEffortCompute():int 和 completePctCompute():double。

(6)R1.8 项目任务选择：Project 类应具有属性“当前被选择任务”，不过，该属性是否应该为 Project 类的一个实例变量还可以进一步讨论，在这里主要是为了将需求中的信息在模型中进行体现。当然，若开发者经验比较丰富，则这个变量可能会在初始迭代中就会被识别和设置。

(7)R1.10 对其他项目的依赖：Project 类应具有 addPredecessor(Project):void 方法。

(8)R1.11 工作量改动的验证：子项目与项目之间的归属关系在类图中是通过 Project 类与自身的关联“子项目”进行体现的，它描述了项目与子项目之间的对应关系。其中，“parent”端基数的表示为“0..1”，意味着每个项目归属于一个或零个其他的项目。如何将设计中的关联关系通过实际的代码实现，将在第 6 章中进行详细说明。

Project 类加入方法 testEffortModification(neweffort: int): bool，这里不仅给出参数类型，而且一同给出参数名字。进而，还可加入方法 testEffortModification(ProjectTask, int): bool。

Project 类还应提供一个方法 effortToAllocate(): int，用来计算剩余的待分配工作量(计划工作量减去已分配的工作量)。

(9)R1.12 工作量检查：这个需求只是对 R1.11 工作量改动的验证给出的方法在业务实现细节上的具体要求。

(10) R1.13 工作量改动失败：ProjectMan 类应能通过方法 inconsistentUpdateNotify (reason: String): void 来提示出现的问题。由于此功能与用户交互相关，因此应该在这里做个注释，即后续可能会由于统一加入了图形界面类和对交互消息的处理而导致对该方法的调整。

通过以上对需求的进一步分析和对方法的识别，可以得到图 5.7 所示的类图。方法在每个类的描述中，位于属性之下的矩形区域内。方法前的加号“+”表示方法的可见性为 public，其含义是可由外部类进行调用，与属性类似，可选的可见性还有 protected (#)、private (–) 和 package (~)。为了简洁起见，图中省略了初始类图中的构造型、关联关系名字和方向等细节信息。

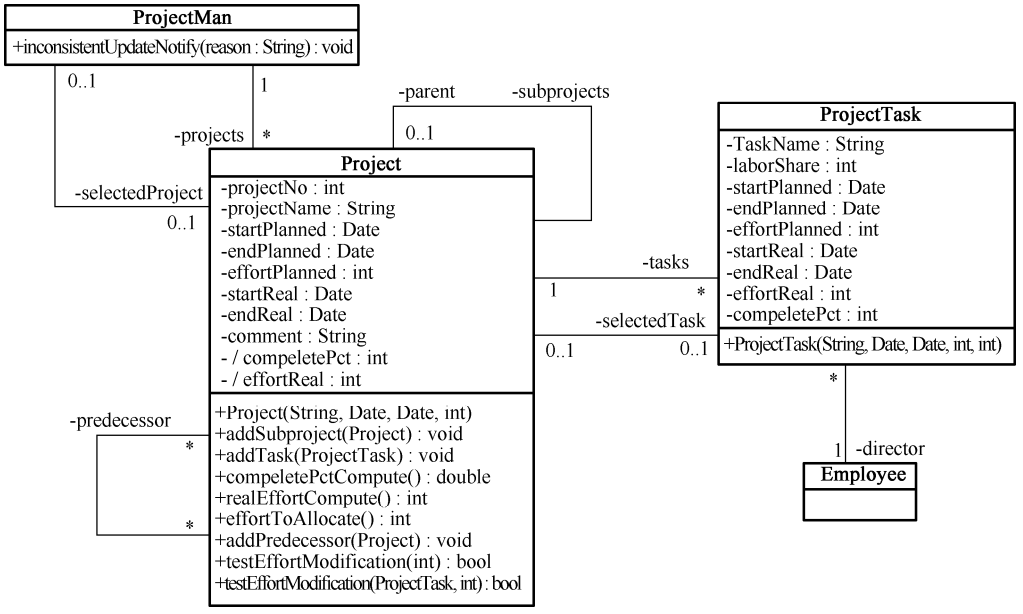


图 5.7　加入方法后的类图

在上下文情境较清晰的情况下，UML 允许细节信息的省略。例如，同样的类在 UML 中有不同的表达方式。图 5.8 总结了几种常见的方式。基本上，任何类都可以将所有信息隐藏起来，只保留名字，这是一种非常简化的表示方法。图 5.8 中表示的都是同一个 Point 类，当人们关注的是类图的概貌及类之间的关系时，对类的细节可能并不留意，这时在不同的上下文中一般只保留类的部分关键的信息，因此也就有了不同抽象级别上的不同表示方式。这与面向对象的开发阶段往往也能够对应起来：随着开发的推进，类的细节信息也会逐渐地丰富起来。

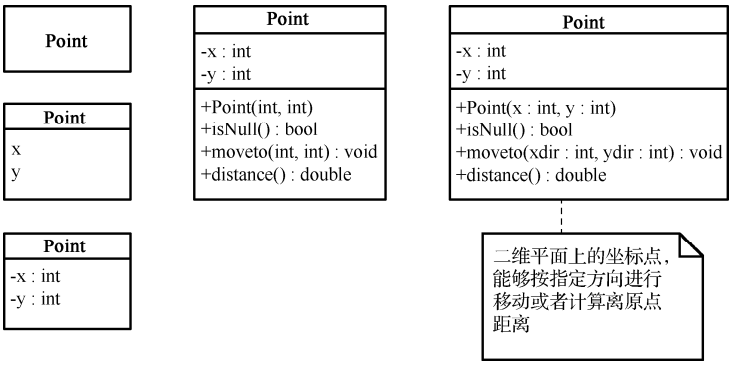


图 5.8　类的不同表示方式

5.2.2 设计优化

在进一步的迭代中，可以对现有的分析模型继续优化，以求获得更佳的稳定性和可理解性。可理解性对于后续的开发活动至关重要，只有分析设计思想的充分传递和理解，才能在结构上获得更趋于稳定的效果，进而保证编码阶段的质量。优化时，应该删掉那些没有方法和实例变量的类；对于比较复杂的类，要对其功能的独立性进行考察，确认是否可以按照功能的不同对其进行拆分，并使得拆分后的类在功能上具有不同的侧重点。

若某些类具有很多的相似之处，则可以通过使用一个上层类对它们进行泛化。在这个例子中，项目类 Project 和项目任务类 ProjectTask 具有很多相似之处，将这些相似内容提取出来，构成两者的共同父类项目组件类 ProjectComponent，如图 5.9 所示。ProjectComponent 类的子类 Project 类和 ProjectTask 类中会自然具有父类的所有实例变量和其自身特有的实例变量。ProjectComponent 类的实例变量前使用了“#”符号，表示可见性为 protected，意味着在该类内部及其子类中都可直接访问，对于其他类，则与可见性 private 一样，不能直接访问。

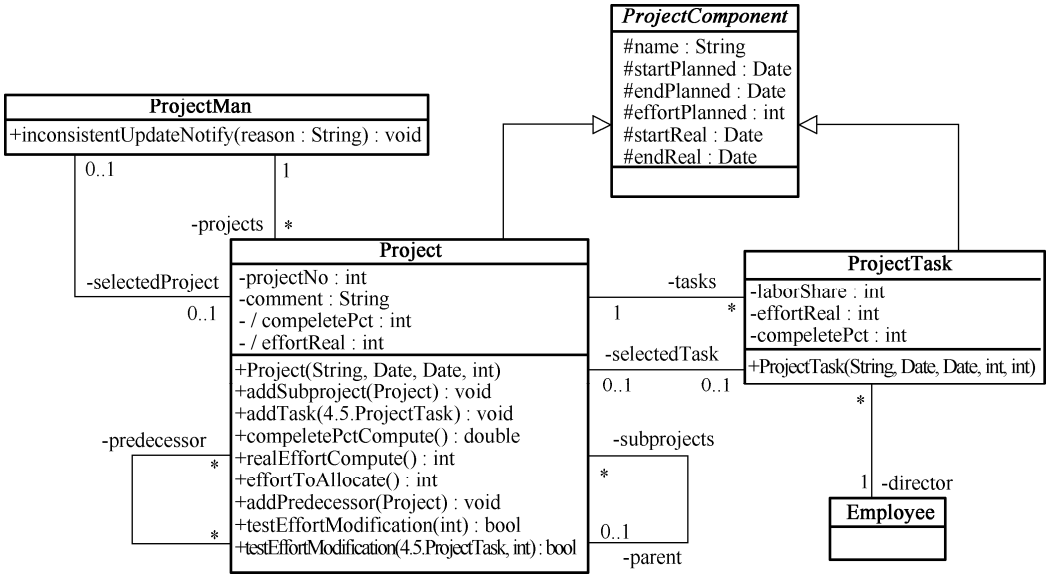


图 5.9　带有泛化关系的类图

另外，ProjectComponent 类的名字使用了斜体字，表示这是一个抽象类，即不能从该类直接产生对应的实例对象。抽象的 ProjectComponent 类因此可以作为变量的一种类型来使用，并由此借助多态特性，避免对象之间的硬绑定，实现松散耦合的效果，从而使得整体结构更加稳定。

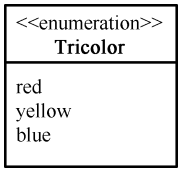


图 5.10　枚举类的表示

另外，如果一个变量的取值是某个有限集合中的数据，如“红色(red)”“黄色(yellow)”“蓝色(blue)”三原色，就应该使用枚举类型，而不是直接使用 String 或 int 等类型。枚举类型可以使设计和代码简单易懂，同时提高取值的可控和安全性。图 5.10 所示的枚举类型具有一个构造型<<enumeration>>，实体类中该类型的实例变量取值应为包含的某个可能的枚举值。

5.3 补充和确认

当初始版本的分析类图完整地构建出来后，需要确认需求中的所有信息是否在模型中都得到了体现而没有遗漏。为了完成这样的验证，可以使用 UML 中的顺序图对需求场景中涉及的不同对象之间的交互过程进行建模。

类图在 UML 中是一种静态图，因为其描述了系统功能和结构的侧面，而基于类图的顺序图，可以设计对象之间的动态交互过程，描述特定场景中对象之间的过程调用顺序和关系，通过顺序图，可以对复杂业务场景进行辅助分析，补充类图中的功能说明，确认是否实现了活动图中描述的功能性需求。

5.3.1 顺序图

顺序图的基本构成如图 5.11 所示，首先在顺序图的上端将场景中所涉及的对象横向罗列出来，构成对象的横轴；顺序图从上到下表示时间的延续，构成了时间的纵轴。每个对象垂直向下延伸的虚线称为生命线，在生命线上可嵌入控制焦点，用以表示对象当前为激活状态，如等待某个结果返回。由一个 object1 指向 object2 的实心三角箭头表示同步消息，通常带有一个结果的返回，同步消息表示 object1 在调用方法后处于等待状态，直到 object2 将结果返回；另外，还存在异步消息，object1 不用处于等待状态，描述上通过一个简单箭头进行区分。

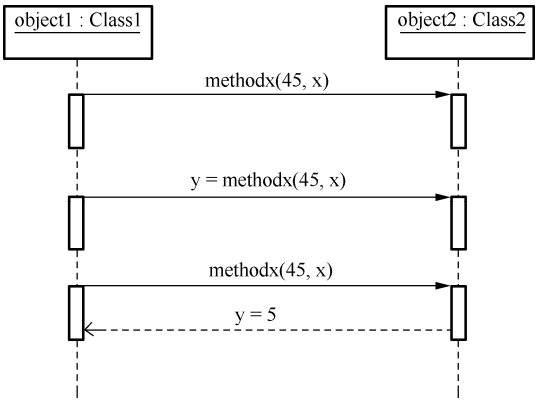


图 5.11 顺序图的基本构成

此外，消息中的参数也可根据情况进行指定。图 5.11 针对顺序图中对象间的同步消息给出了三种不同的描述方法。第一种调用方式具有一个代表控制焦点的矩形框，表示对应对象处于激活执行状态。在这种表示方法中，更能清晰地表示出 object1 调用并等待 object2 方法执行结束并返回结果。接下来的两种方式都明确地表示了计算结果的返回情况。选用哪种表示与具体的要求有关，一般的做法是在分析的早期，可能只给出简单的方法名和箭头方向，在随后的细化中根据需要再逐渐优化。

生命线表示对应对象实例的生存周期，如果对象被销毁，通常在该对象的销毁处使用“×”

进行标记，并且虚线不再向下延伸。图 5.12 中左边描述的是对某对象的创建操作，箭头所指是一个新创建的对象，注意此对象的矩形要低于其他对象，右边是一个对象的删除操作。

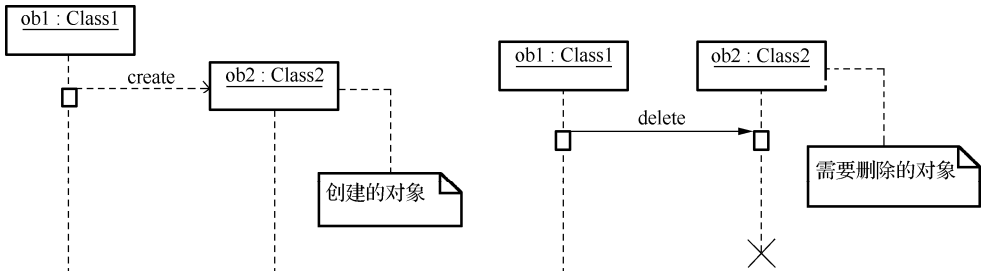


图 5.12　对象在顺序图中的创建和删除

除了顺序图，还可以使用通信图(Communication Diagram)来对对象间动态的交互进行可视化建模。图 5.13 描述了与顺序图在相同交互场景中等价的通信图。通信图中的每个对象与顺序图中的描述格式一样，即矩形内带有可选的对象和类名的形式。对象之间通过实线连接，表示它们之间具有的连接关系，这其实是对应类的关联关系的实例。在连接线的一侧描述了消息的交互信息，交互顺序通过消息前面的数字进行标识。如果在消息内部含有嵌套消息或者又触发其他事件，那么这种嵌套的层次关系可用上层数字后的扩展数位来表示，如消息 2.1 表示消息 2 的内嵌消息，并可扩展多层。

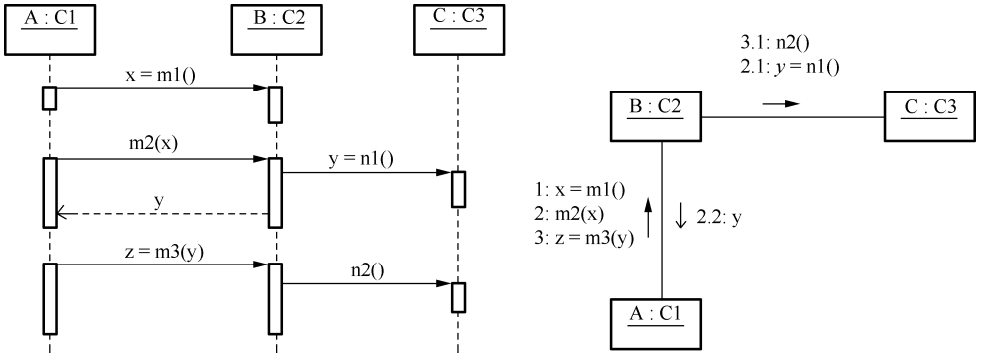


图 5.13　顺序图与等价的通信图

图 5.13 中也给出了顺序图中两种不同的返回值的描述方法对应在通信图中的表达方式。通信图可以很好地用来描述对象的结构和它们之间的依赖关系，更强调交互对象的结构性，但如果涉及比较复杂的交互时，使用通信图会比较烦琐，因此在本书中使用顺序图。由于两种图具有等价性，因此使用哪种模型只是一种习惯而已。

UML 2.0 以后的版本中，顺序图不仅可以用来描述顺序的过程，而且可以描述多种带有选择结构的逻辑。这些逻辑结构通常使用一种矩形框进行表示，并在左上角给出逻辑结构的含义。图 5.14 中给出了三种常用的逻辑结构，其中“opt”表示此部分为可选的内容，表示在满足方括号内条件的情况下，对应部分会被执行，否则跳过；“alt”是对多分支的条件进行选择，在矩形框内各分支彼此之间使用虚线进行分割，每个分支都对应一个条件，而且彼此排斥，若条件都没有满足，则 else 部分会被执行；“loop(start, end, condition)”可用来对循环结构进行定义，这里必须清楚地给出循环执行的参数，如循环次数和结束条件。

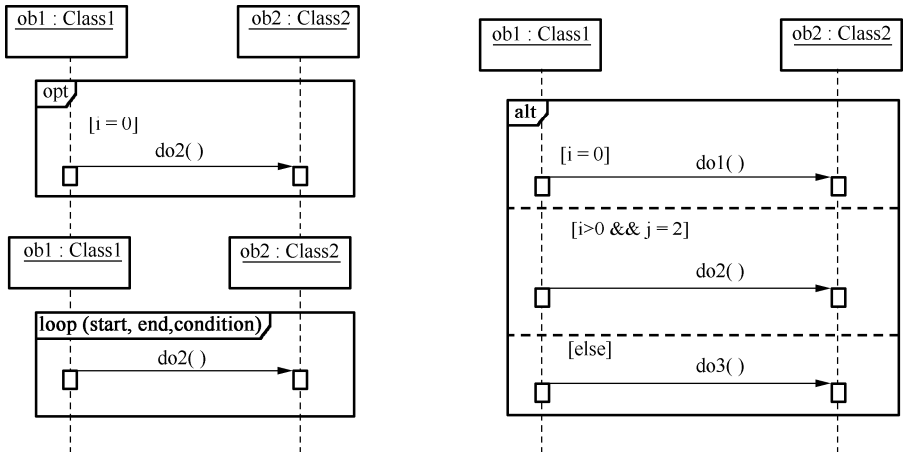


图 5.14　顺序图中的逻辑结构

5.3.2　场景模拟

使用顺序图对活动图(对应着用例规约)进行确认的方法是对活动图中的每个场景使用一个顺序图进行模拟，即对实际的业务流程进行场景模拟。这个过程如图 5.15 所示，活动图中描述了 3 个业务流程，分别使用实线、长虚线和短虚线进行了标识，确认的目标是确保每条活动图中的边都被执行。对于每个可能的执行流程，都寻找一个顺序图与之对应，使其具有相同的功能描述。如图 5.15 右边给出的顺序图分别对这三个子过程进行了模拟。当然，单一的子过程不必多次在每个顺序图中重复模拟，可以对一个或多个动作创建子图。

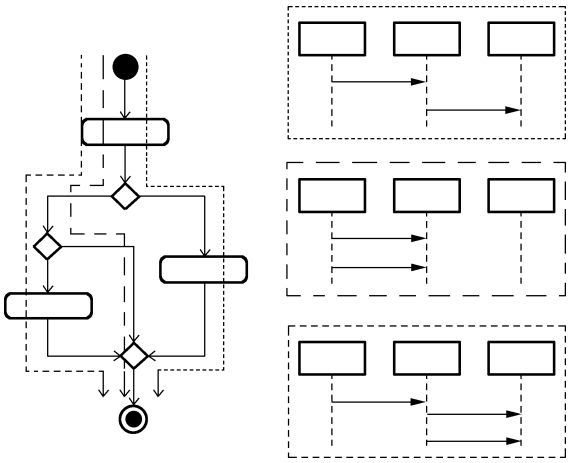


图 5.15　业务流程场景模拟

使用顺序图可以方便地对业务流程进行场景模拟，包括对象的创建和销毁过程，确保这些功能性需求在分析模型中得到了正确的体现，而需求中的非功能性需求则主要是在系统架构层面体现出来的。分析类模型及其衍生模型和顺序模型的创建也是一个迭代过程，在不断加深对需求理解的同时，丰富和调整设计，使设计日趋完善。这个过程没有固定的思维和规则可以照搬，需要开发者具备基本的软件素质和创造力，是日积月累的开发经验的沉淀和升华。

下面以“软件项目管理系统”为例，讨论使用顺序图的实际建模过程。在第 3 章的图 3.15 中“项目信息编辑”的活动图中，要求必须具有对新项目的创建功能，包括对应的子项目和项目任务的创建。这个场景可模拟为图 5.16 所示的顺序图。首先，项目管理对象 ProjectMan 创建了两个新项目 pa 和 pb，然后 ta 和 tb 两个项目任务被创建，项目 pb 作为子项目加入 pa 中，ta 和 tb 作为项目任务分别加入项目 pa 和 tb 中，最后，对项目 pa 做了简短备注。

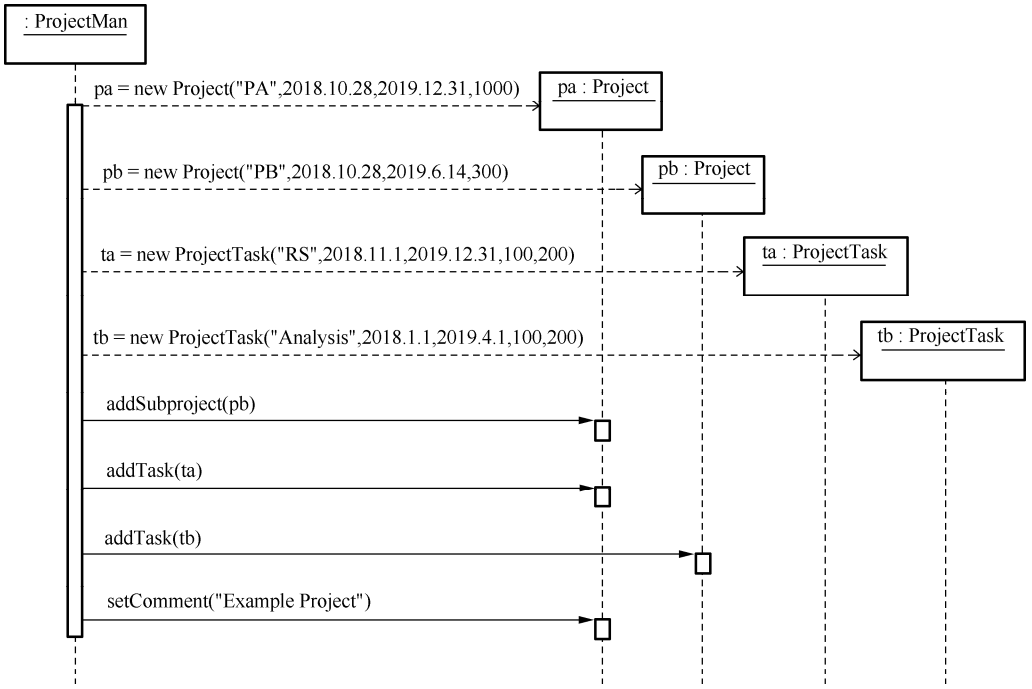


图 5.16　新项目创建的顺序图

这个场景中的项目管理对象 ProjectMan 从何而来？它可以是包含在图形边界中的一个对象。场景内会存在一些外部对象，它们与场景内的对象有交互作用，可能暂时无法获知它们的准确信息，但确有存在的必要，因为它们可能是业务流程的驱动者。这时可在顺序图的最左边使用一个“Extern”对象将其表示为一个附加对象。图形边界中的类似对象一般是在后续实现步骤中逐渐补充进来的，由于所用的编程语言不同，它们的具体类型和方法都不尽相同。

图 5.17 给出的示例描述了通过外部类对流程进行的驱动。外部 Extern 类使用了“角色”的图标代替，清晰地表达出这个类不属于该用例场景内的实体类。

接下来考虑“项目修改”的业务需求，这里重点关注对项目工作量属性的修改。根据设计，若项目之间存在继承关系，则通过成员变量 parent 关联，因此该用例可能会涉及多种情况。如果只是提高项目对象的工作量值，对于其包含的子项目或项目任务来说就不会有什么问题，因为只是增加了更多的工作量待分配；但是，如果这是一个子项目，就要结合其父项目进行严格的验证，以确定修改的合理性；另外一种情况就是该项目的工作量减少了，则也要对其子项目和所含任务的工作量进行一致性检查。

以上的业务分析需要在该用例对应的顺序图中进行表现，图 5.18 描述了对项目设置新的工作量值 newvalue 时进行检查的流程。首先比较新值和原值的关系，原值是在项目对象的实例变量 effortPlanned 中存储的。如果新的工作量值较小，Project 类计算已经分配的工作量，

这是通过方法 effortToAllocate()实现的。计算的过程在这个顺序图中不做展开，可以在这里或者通过另外一个顺序图详细描述。

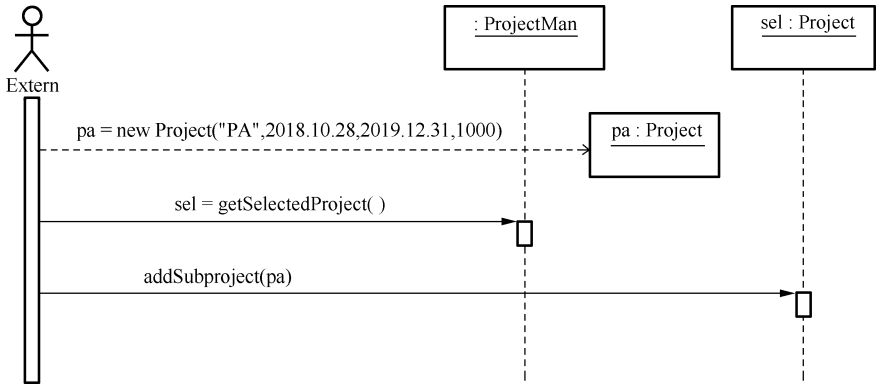


图 5.17　外部驱动的项目创建流程

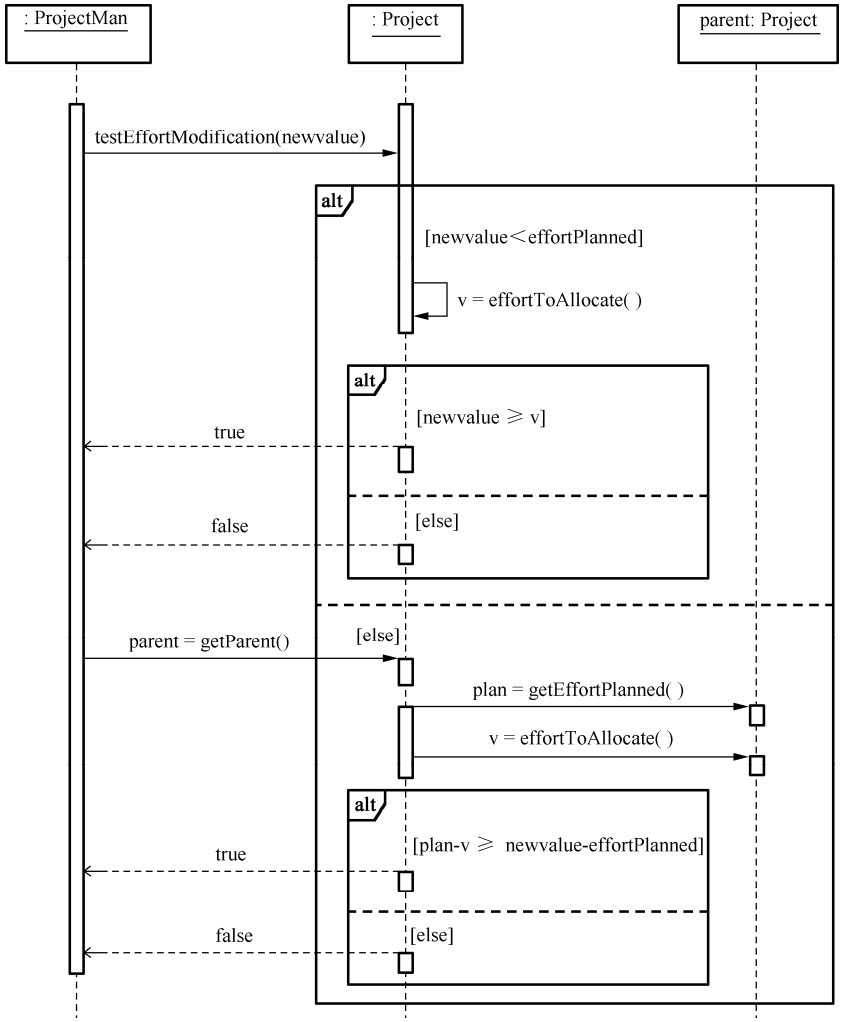


图 5.18　项目工作量的修改

计算出已分配工作量后，进一步通过一个选择结构再次判断该新值是否大于等于这个已分配的工作量值，若满足，则意味着符合数据逻辑关系，可以安全地对工作量进行修改，否则不行。接着，回到第 1 层的判断，即如果新的工作量值大于项目计划工作量，必须首先取得其父项目的计划工作量 plan 与已分配的工作量 v。若父项目待分配的工作量值(plan–v)大于等于该项目工作量的增量(newvalue–effortPlanned)，则允许修改(返回 true)，否则不允许。

类中经常会存在一种方法，它们主要用于类内部的计算。在顺序图中的描述形式是从某对象的生命线出发又回到同一生命线的消息，即自调用，因此可以将其可见性设置为私有或受保护的类型，如图 5.18 中项目对象 Project 对自身发送的 effortToAllocate()消息。

与项目的修改流程类似，对单个任务的工作量值也可以进行修改，但修改逻辑有所不同，对任务的修改不涉及下层子任务或子项目，因此当任务工作量减少时，并不需要做额外的判断。任务工作量修改的场景描述如图 5.19 所示，其中 ProjectMan 类会将某项目任务的修改请求先发给它的 Project 类，接着 Project 类查询出该任务的计划工作量值 plan，然后计算项目中已分配的工作量值 v。若项目中待分配工作量(effortPlanned–v)大于等于该任务工作量的增量(newvalue–plan)，则该任务的工作量允许修改。需要注意的是，这个增量可能为增加，也可能是减少，但在这里不需要区分，可以统一处理。

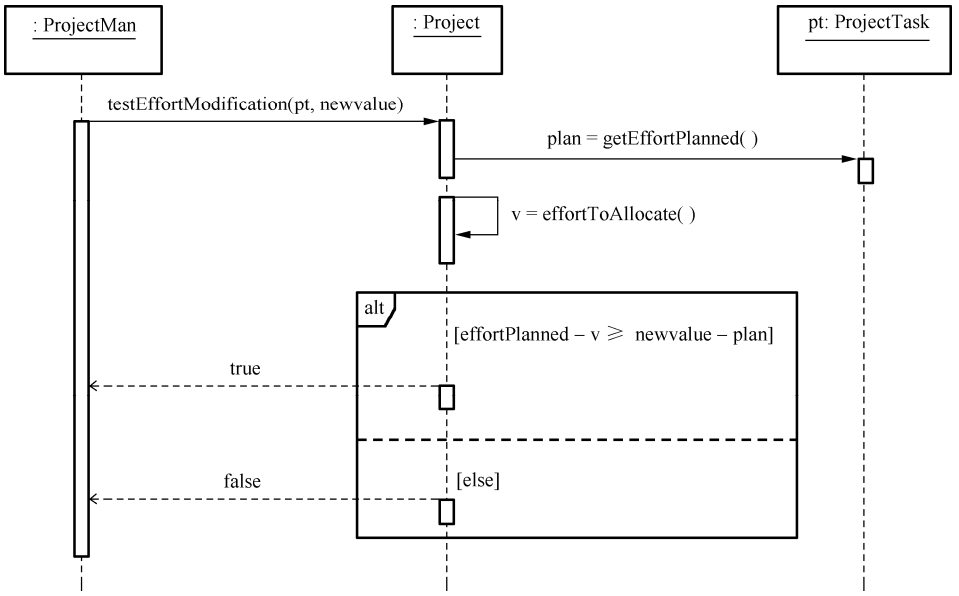


图 5.19　任务工作量的修改对应的顺序图

对于以上两个顺序图中包含的计算已分配工作量的过程 effortToAllocate()，并没有给出详细解释，这个过程类似图 5.20 中关于计算项目完成进度的流程，因为工作量的计算过程同样也涉及子项目和任务。具体地，为了计算整个项目的完成进度，需要逐个确定每个子项目的完成进度，这里使用的是递归的方式，因为在设计上，项目与子项及任务构成了一个树状的结构，子项目与项目的处理方式也是相同的。接下来，确定每个任务的完成进度及由这些返回的结果，确定项目总体的完成进度。图中的消息返回都没有显式地给出，因为这里的重点是强调业务整体上的处理流程，验证分析模型的完备性和合理性。

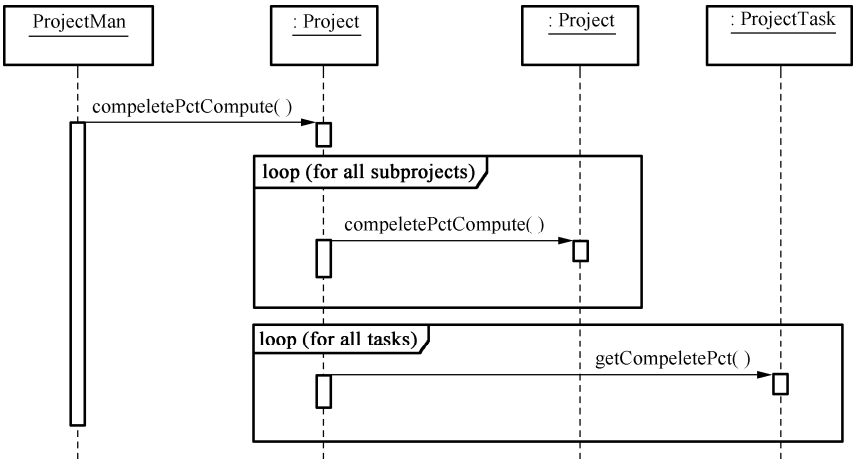


图 5.20　计算项目完成进度

5.4　界面类设计

界面类设计相对独立，其设计的基本要求是：通过界面使得模型中含有的业务类的某些部分对外部可见，如用户通过人机交互界面对某些业务内容进行修改或访问。借助顺序图来描述与系统的交互过程比较方便，而且界面类的建模较具体，与物理实现比较接近，所以界面类的设计与实体类设计比较起来，一般可推迟进行，通常可直接应用现成类库中的模型元素，如 Java 的 Swing 类库，不同的类库对整体的类设计方案也会有影响。

如果要对现有的实体类模型扩展对应的界面描述，直接的方法就是对每个实体类补充一个对应的接口。举例来说，对于 Project 类，可为其设置一个 ProjectMask 界面类，对外提供项目创建和修改等业务封装。进一步，可以使用一个控制类，如 GUIControl 类，其作用是响应用户的功能选择，控制当前处于活动状态的界面类。界面类中含有的方法一部分来源于对应的实体类，在实现中，这些界面对象会获得并使用输入的数据，然后调用对应的类方法驱动业务。另外，界面类中还有一部分方法直接来源于需求规格说明书中文本形式的需求，如第 3 章的图 3.17 中类型 2 的需求正是针对用户交互的描述。这样的界面类可看成是边界类的一种，因为它们是系统与用户之间存在的边界。另外的边界类对应类型 3 的需求，是系统与其他外部系统的交互边界。



图 5.21　边界类和控制类

与实体类类似，边界类和控制类在 UML 中也有对应的构造型<<boundary>>和<<control>>，表示上也有便捷的图形符号，如图 5.21 所示。

图 5.22 是“项目编辑”用例对应的初始设计方案。可以注意到界面类的方法中没有参数，只是一个对功能性需求的抽象说明，这是因为它们需要的参数一般是在外部通过界面元素读取并初始化的，如文本输入框。具体的界面布局类的说明在这里省略了，因为它们通常与具体的编程语言相关，而且对于设计类图来说，主要目的是加入对界面或接口规格的说明以完善模型。在这个过程中，一些必要的新的方法还可能被识别和加入进来，如在 ProjectMan 类中加入一个用来返回指定位置的项目对象的方法 getProjectAt(position)。

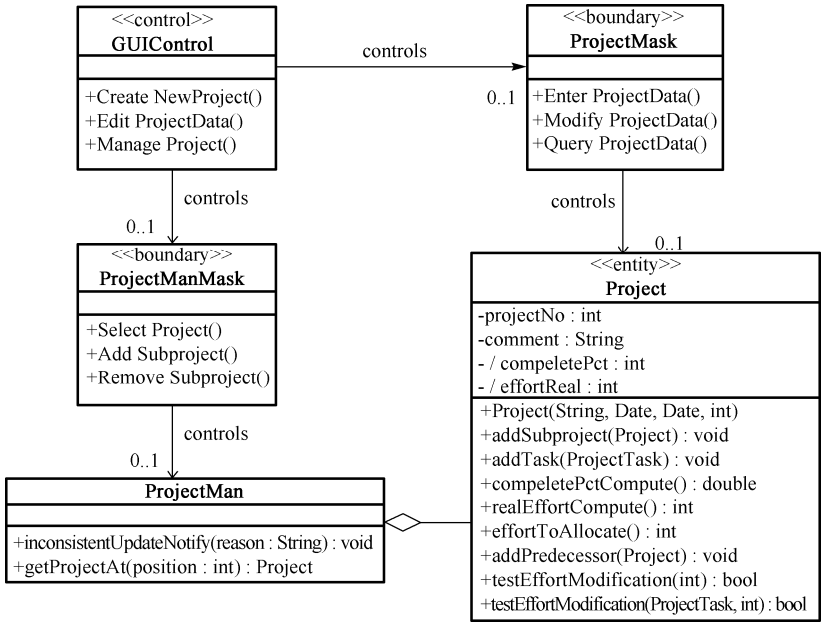


图 5.22　“项目编辑”用例对应的初始设计方案

图 5.23 通过顺序图描述了一个带有子项目的项目对象被创建的过程。因为界面类的方法没有非常具体地给出，所以这里的消息部分使用了文本形式进行描述，主要用于设计思想的说明及辅助理解，这在分析和设计早期阶段是很常见的。

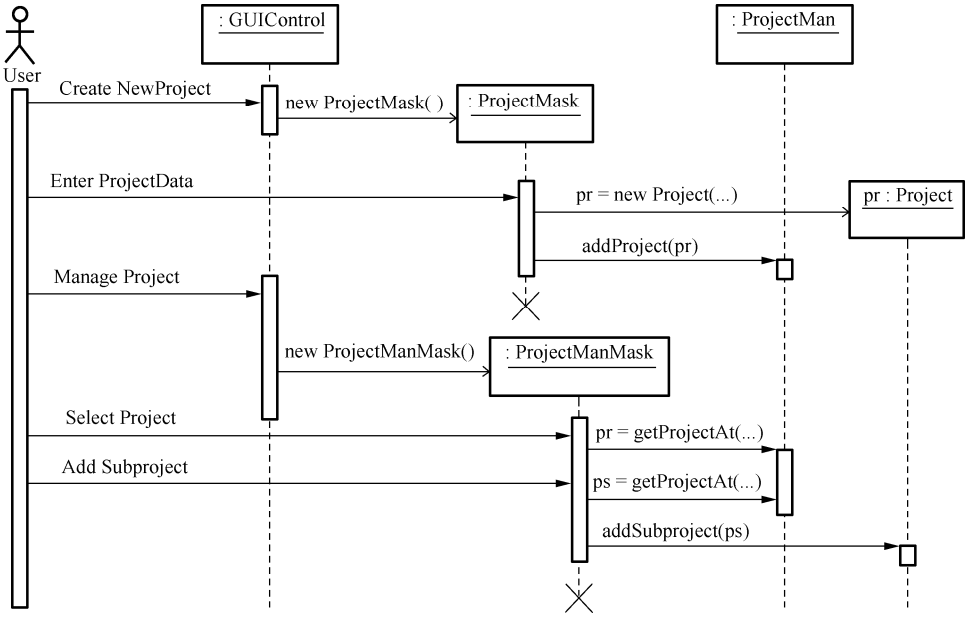


图 5.23　带有子项目的项目对象创建

在该例中，每个类都配有一个界面类。对于管理类或控制类来说，这是必要的，因为它们都分别需要通过界面来响应对业务实体的创建和访问等请求。在很多实际的业务中，一个界面类可能会涉及多个业务实体，也就是说，如果所有的业务驱动都围绕控制管理类展开，这样

就可以减少场景中参与的类的数量，也就简化了模型的消息，从而有助于对系统整体的理解和维护，这样也就不需要为每个实体类设置界面类了。

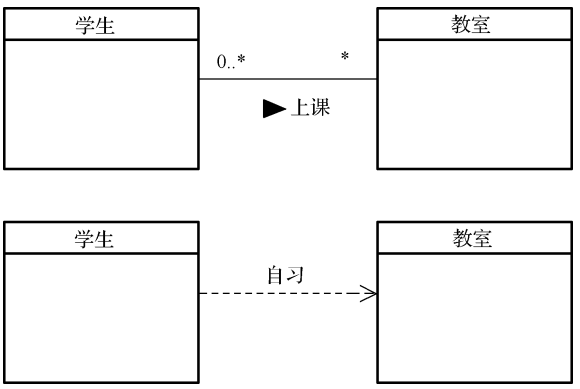
除了界面的功能分析，界面的布局也属于设计的任务。这里涉及除功能以外其他更多的方面，如系统的易用性及用户体验等，图形设计需要专业的团队，但无论如何都要基于对需求深入理解的基础上，才能更有效地展现和实现预期的功能。第 10 章会专门讨论与界面布局设计相关的内容。

5.5 习题

1．针对下述描述建立类模型，画出该系统的分析类图。

某软件公司下属的部门分为开发部门和管理部门两类，每个部门由唯一的部门名字确定。每个开发部门可开发多个软件项目，每个管理部门承担公司的若干项日常管理工作。公司的员工分为经理、工作人员和开发者三类。开发部门包括经理和开发者，管理部门包括经理和工作人员。开发项目时，每个项目只能由一位经理主持，但一位经理可主持多个开发项目；每个开发者可参加多个开发项目，每个开发项目也需要多个开发者参与。

2．以下两个类图的含义有何不同？



3．针对第 3 章习题 2、3、4 的描述，在它们用例图的基础上分别给出各自的分析类图。

第6章 代 码 生 成

前面的章节介绍了从需求文本出发产生初始的分析和概要设计类图的过程及通过对场景的动态模拟来对类图进行验证。在这个验证的过程中，通过使用顺序图来描述既有场景内参与对象间的交互，验证的过程实际上也是一个对现有业务的深入理解和对设计方案不断完善的过程。在接下来的步骤中，我们要考虑这些设计方案向实际运行的方式的转变过程，即由概要设计产生出对应的程序代码框架的过程。

工程化的开发设计方法会导致程序代码具有更好的可实现性、可维护性、可修改性及可扩展性。具体的优化策略会在后续的章节中介绍，本章首先对从类图到可代码的基本转换过程进行概要说明，然后对其进一步细化。

6.1 逆向工程与 CASE 工具

现今的软件开发在实践中都离不开软件开发环境的支持，软件开发环境是指支持软件开发的工具及其集成机制，用以支持软件开发的过程、活动和任务，为软件的开发、维护及管理提供统一的平台，即计算机辅助软件工程(CASE)。不同类型的项目及其软件开发环境的构成不尽相同，在工具的选择上也会有较大的差异。

大型软件开发厂商如 IBM 开发的 Rational 和 Together 工具，提供了从需求获取到可交付的代码过程中的所有相关工具和完整过程支持的套件。这里所说的套件，是指其中包含的工具涵盖了开发的主要阶段，它们各有所长、相互协调，共同完成开发任务，但彼此之间并非紧密依赖，缺一不可。例如，并非强制要求使用需求跟踪管理的工具。工具的选择可能会对开发过程的选择产生一定的影响，因为不同的工具组合对开发过程的支持能力会有所不同。

另外可选的方法是在每个单一的开发步骤中引入不同的工具支持。这需要将现有的开发工具与新的工具进行集成，以支持未来的开发步骤。这样做通常会导致工具间的接口问题，因为工具之间的数据格式可能并不兼容。

在软件开发环境中，一种常用的工具框架是集成开发环境(IDE)，它们有很多开源产品可供使用，如 Eclipse 主要为 Java 项目提供开发支持。Eclipse 的优点是能够通过插件的安装扩展其自身功能，使更多的工具集成到环境中，类似的工具还有 Netbeans 等。

实际上，CASE 环境的搭建与开发过程的选择具有很强的依赖关系，其中较重要的一点是要考虑开发过程中对各种“变更”的管理方式，这里存在两种较为极端的情况：

(1)需求分析、概要设计和详细设计过程只进行一次或者迭代增量式地进行。当编码开始后，对已经完成产品的修改只在程序代码中体现。需求分析、概要设计及详细设计文档不做更新。

(2)对每个改动的意愿，都要经过完整的分析、概要设计和详细设计流程，所有的改动都需要在所属的文档及代码中对应修改，并保证它们的一致性。

开发过程的管理与实际的项目相关。例如，若项目需要长期维护且不断扩展功能，则第

(2)种方法比较适合；若项目结束后不需要长期维护，则第(1)种方法较为合适。当然，它们的折中方式在实践中也会经常采用，具体方式的选择对未来的开发和维护成本有着直接的影响，但合适的 CASE 环境会使开发活动变得事半功倍。

从类图向代码实现的转化过程中需要引入正向和逆向工程的技术方法。正向工程是从类图出发生成程序代码的框架，进一步的开发可直接在代码上进行。逆向工程是将代码的修改反向映射回类图的设计中，从而在设计与代码实现之间保持一致性。逆向工程的一种特殊用途是完全通过代码生成进行设计，在原始设计缺失时，可以起到对代码快速理解的作用。

逆向工程使得所有的开发可在 CASE 环境中同时开展，并使设计与实现之间相互对应。但是，这些工具之间可能并不能完全地无缝集成，尤其是当引入了某些高效或者独立的开发工具时，它们可能会阻碍逆向工程的进行，使模型和代码之间的紧密性大打折扣。因此，对于工具和开发方法的选择，需要充分考虑各种开发因素并权衡利弊。

6.2 单个类的实现

本节主要介绍从类图到程序代码框架的生成过程，这是由设计到实现必然要经过的环节。我们在不断对业务进行理解的过程中，对各种分析模型和设计模型进行完善，最终将它们“翻译”成对应的代码，这在实践中也是开发系统原型时常采用的方法。在大型项目的设计过程中，还会不断对已有的初始模型进行细化，例如，补充描述行为的状态图和更为精细的业务规则，项目不能将其本身作为子项目进行添加，要在模型中进行体现。

首先，针对单个类的实现进行说明，一个类图如果可以成功翻译为代码，类模型中的内容必须要完整。类中需要包含的信息如下。

(1)每个实例变量，需要指定其类型。

(2)每个方法中的参数和返回值，需要指定其类型。

(3)每个关联关系，其关联类型、导航方向必须说明。

图 6.1 给出了类图中实体类 Employee 的完整表示，它所有的 get 和 set 方法也都显式地给出。其中某些方法和属性下面带有下画线，代表类变量(静态)和类方法(静态)。类变量表示它用来对类范围的某个属性进行说明，而不是类的每个对象。每个对象可以访问其对应类中的类变量，但对其改动将作用于该类的所有对象，因为类的类变量在内存中只存储一份，在所有的对象实例中共享。通过类方法，可以实现对类变量的访问，但类方法不能访问一般的实例变量。类变量在编译时预分配存储空间，不需要使用任何方法在运行时动态分配。对于类方法的调用，在大多数程序设计语言中，可直接使用以下形式：

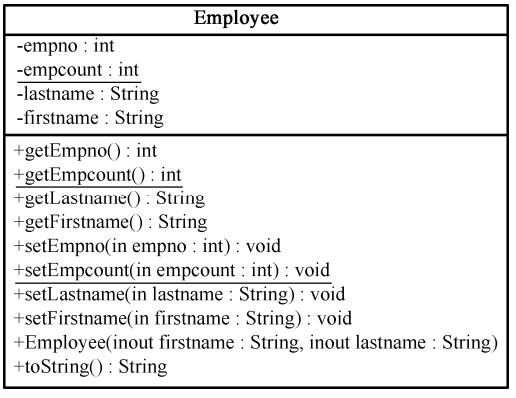


图 6.1 具有类变量和类方法的类

```
<ClassName>.<StaticMethodName>(<Parameters>)
```

对于类变量的使用需要慎重，因为它会破坏面向对象的封装性原则。类变量的实质是一

个静态变量，对于该类的所有实例，都可以共享访问，从而增加了这些对象之间的耦合性[①]。类变量和类方法通常用在常规任务中，如记录文件的存储路径或者常规的数值计算、加密等。习惯的做法是将这些常规方法以静态的形式在一个工具类中封装，不含有其他实例变量和方法，所有外部类可方便使用该工具类，因为不需要去创建该类的实例。典型的例子就是 Java 中的 Math 类，其中包含了绝大多数基本的数值计算功能。另外，利用类变量在实例之间的共享特性，可以用来记录一些全局信息，如生成唯一的员工编号等。

此外，类中方法的参数前使用了“in”关键字，用来表示参数在方法内部是只读的，不会被修改。同样，“inout”也是可以在参数前使用的关键字，表示该参数在方法的处理过程中会被访问，也可以被修改，而且在方法结束后对该参数的修改结果可以保持，即可以被外部接收到。关键字“out”表达的含义是该参数只能作为方法内部计算结果，也就意味着调用该方法时，out 参数可被赋予任何值(哑值)，对于方法的内部计算不起任何作用，方法结束后该参数记录并保持计算结果。在很多类图中，对参数的形式可能不会描述得很细致，大多会省略。Employee 类的代码实现如代码 6.1 所示。

代码 6.1　Employee 类的代码实现

```
public class Employee{
    /**
    * @uml.property name="empno"
    */
    private int empno;
    /**
    * Getter of the property <tt>empno</tt>
    * @return Returns the empno.
    * @uml.property name="empno"
    */
    public int getEmpno(){
        return empno;
    }
    /**
    * Setter of the property <tt>empno</tt>
    * @param empno The empno to set.
    * @uml.property name="empno"
    */
    public void setEmpno(int empno){
        this.empno = empno;
    }
    private String firstname = "";
    public String getFirstname(){
        return firstname;
    }
    public void setFirstname(String firstname){
        this.firstname = firstname;
```

① 这里是公共环境耦合，是一种耦合程度比较高的情况。

```
    }
    private String lastname;
    public String getLastname(){
        return lastname;
    }
    public void setLastname(String lastname){
        this.lastname = lastname;
    }
    private static int empcount;
    public static int getEmpcount(){
        return empcount;
    }
    public static void setEmpcount(int empcount){
        Employee.empcount = empcount;
}
    public Employee(String firstname, String lastname){
        this.firstname = firstname;
        this.lastname = lastname;
        this.empno = Employee.empcount++;
}
    @Override
    public String toString(){
        return empno + ": " + firstname +  " " + lastname;
    }
}
```


代码 6.1 是在设计类的基础上较为完整的代码，并可直接在实际项目中使用。在代码的转换中还补充了一些注释内容，这些注释包含两类：一类是在设计类图时给出的，由 CASE 工具自动生成；另一类是 CASE 工具内部使用的注释，用以在类模型和代码之间起连接的作用，为逆向工程提供支持。为简洁起见，这里只给出了属性 empno 的注释，其他的实例变量也是类似的。开发者最好不要修改自动生成的注释，否则可能会丢失参照信息，从而失去逆向工程的能力。

在代码转换过程中，还采用了某种编码风格，如首先给出实例或者静态变量的定义，然后紧跟着是对该变量的 get 和 set 方法。代码的结构风格在一些 CASE 工具中支持定制，并可以实现不同的代码风格间的自由切换，以便开发者使用习惯的方式对代码快速组织。

当然，通过 CASE 工具生成的代码框架通常较为简单，没有过多的业务代码，这就需要开发者在此基础上补充具体的业务逻辑，如代码 6.1 中阴影标记的部分就是在生成后的代码框架上补充的代码。对于较复杂的计算业务，只有在设计中给出了详细的业务计算逻辑后，才有可能实现自动的、完整的代码生成。业务越复杂，代码能够自动生成的可能性就越低，对于这部分需求，还需要工具的开发者为此付出大量努力。

6.3 关联关系的实现

通过关联的定义，明确了类与类之间存在的静态关联，这种关联的实现最终体现为类中

增加的实例变量，用以保证对关联对象的持有和保持。这些“关联变量”存在的具体形式依赖于关联的具体类型。

在类模型构建的初始阶段，类与类之间的关联关系往往只是简单地通过一条实线来表达。在接下来的迭代中，关联关系的多重性会被添加进来，表明参与关联的对象间在数量上的对应情况。同样，关联关系的导航方向及角色信息等都会逐渐地完善。图 6.2 描述了关联关系的这些详细内容及其表示。

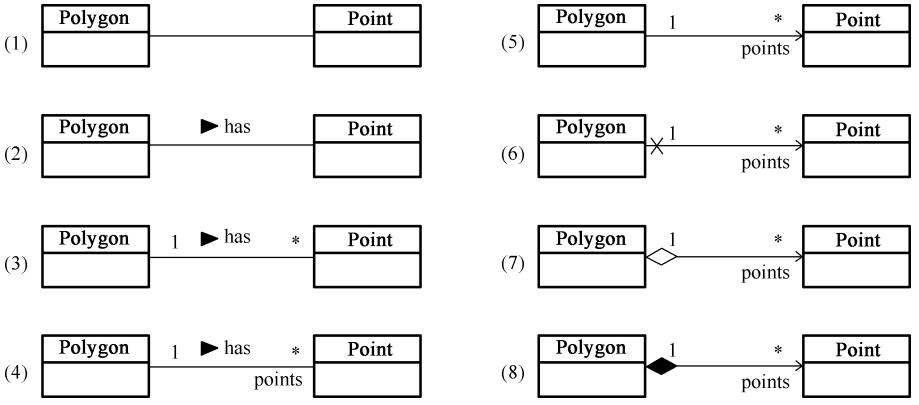


图 6.2　关联关系的不同表示方法

从上至下对该图中的关联关系解释如下。

(1) 多边形类 Polygon 和点类 Point 存在关联关系。

(2) 多边形由点构成。在表示关联关系的实线上写明了该关联的名字，对应的那个实心三角的箭头方向表示了该关联名字的方向。这个关联的名字信息将不会出现在转换后的程序代码中。

(3) 一个多边形是由任意多个点构成的，一个点只属于一个多边形。在此情况下，多个多边形对象之间不存在共享的点对象。若不同的多边形存在交集，即它们具有的某些点含有相同坐标值，则实际上是两个具有相同坐标属性值的不同点对象。

(4) 每个多边形含有任意多个点，这些点通过多边形类的一个(集合类型)实例变量 points 进行管理。每个点只属于一个多边形。

(5) 每个多边形含有任意多个点，这些点通过多边形类的一个实例变量 points 进行管理。该关联关系存在由多边形类至点类的一个简单箭头指示的方向，表示只能从多边形获知其包含的点信息，而不能从点获知其所属的多边形信息。这个获知对方信息的方向称为导航方向(Navigation)。多重性“1”在这里对代码的生成没有太大作用，因为点并不感知其所关联的类的存在，但这个数字“1”却是必要的，因为它描述了一种约束条件。对于开发者来说，要在代码中采取措施，确保每个点对象只属于一个多边形对象。导航方向对代码的未来实现具有影响，需要在代码中进行体现。

(6) 为使前述关联关系的导航方向更加清晰，可以在关联关系中没有导航能力的一端使用符号“×”显式地表示无此导航方向。

(7) 一个空心的菱形符号表示多边形与点的另外一种关联，即部分与整体之间的弱包含关系——聚合(Aggregation)，表示它们相互之间没有存在层面的依赖性。这意味着如果一个多边

形对象被删除，其所属的点对象会继续存在，并可由其他多边形利用。

(8)一个实心的菱形符号表示对象之间的一种特殊的关联关系，表示部分与整体之间的强包含关系——组合(Composition)。点对象的存在依赖于所属的多边形对象，意味着如果一个多边形对象被删除，其所属的所有点就会被同时删除，即整体与部分之间的一种“同生共死”的关系。

详细设计的目的就是要对每个关联关系进行具体的描述和细化。这里尤其要注意导航方向的确定。一般来说，单向导航方向的代码要比双向导航方向简单一些。

图 6.3 描述的业务含义是：一个 ProjectTask 最多由一个 Employee 承担，通过多重性“0..1”可知，ProjectTask 也可以不分配 Employee。也就是说，在代码实现中，对应类的实例变量可以不赋任何值。在类构造的代码中，可不必对该变量初始化，或者对该变量赋予空值(null)，表示不分配任何存储空间。Employee 类与 ProjectTask 类的关联关系的实现如代码 6.2 所示。

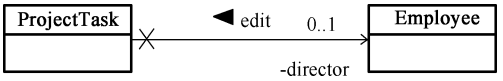


图 6.3　Employee 类与 ProjectTask 类间的关联表示

代码 6.2　Employee 类与 ProjectTask 类的关联关系实现

```
public class ProjectTask {
    private Employee director;
    public Employee getDirector(){
        return director;
    }
    public void setDirector(Employee director){
        this.director = director;
    }
}
```

这段代码在实现上符合图 6.3 的设计，注意对应的实例变量并不需要在声明时赋值，可以通过 set 方法在后期需要时再赋值。另外，实例变量 director 声明后可以一直为 null，但却不能通过任何方法显式地对其赋予 null 值，这是隐含的业务规则。

图 6.4 描述了一个 ProjectTask 对象必须要有一个 Employee 与其对应，这样也就不存在是否可以对该变量赋予 null 的问题。为确保实现以上约定，需要在该实例变量声明时同时赋予初值，按照以下方式实现：

```
private Employee director = new Employee();
```

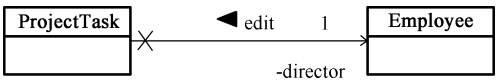


图 6.4　唯一关联的对象表示

同样，在该类的每个构造函数中都要为实例变量 director 指定一个有意义的值。在这种情况下，可以将实例变量声明时的初始化去掉。

图 6.5 中描述了一个 ProjectTask 可以安排任意多个 Employee 与之对应。同样，这些 Employee 也是通过实例变量 director 进行保持的，只是，这时该变量的类型应为某种集合类型(集合类型在 C++中也称 Container)。具体集合类型的选择要根据是否排序或者含有重复元素而定。4 种基本的集合类型如表 6.1 所示。

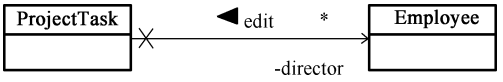


图 6.5　任意多的关联对象表示

表 6.1　基本的集合类型

集合类型	元素顺序要求	元素唯一性要求
Set	不要求	无重复{unique}
Bag 或 Multiset	不要求	允许重复
OrderedSet	要求{ordered}	无重复{unique}
List 或 Sequence	要求{ordered}	允许重复

若在多重性上对元素的最小参与数量有要求，如“2..*”，则在构造该类时需要在实现上注意，至少要在对应的集合变量中加入 2 个元素。若要求“2..7”的多重性，则在实现上选择具有 7 个存储单元的结构(如数组 Array)比较合适。一般，在进行代码实现时，要注意所选的编程语言是否提供了尽可能多的集合类型，这样才会有更多的选择。若想通过一种更简单直接的对象引用方式，如通过对象的某种标识实现对对象的访问，则可以使用另外一种数据结构——映射(Map)。其中的一种映射方式是将元素的标识值与该元素通过 Hash 方式进行对应，达到对元素快速访问的目的。

表 6.1 中的花括号是一种对约束条件的说明方式，可以放在类图中关联关系的一侧进行说明。在基于 Java 语言的开发中，可以使用集合 List 接口类型对应允许元素重复的集合类型，并使用 ArrayList 进行实现。这样的代码片段如代码 6.3 所示。

代码 6.3　使用 Java 的 ArrayList 实现集合类型

```
import java.util.List;
import java.util.ArrayList;
public class ProjectTask {
    private List<Employee> director = new ArrayList<Employee>();
}
```

因为 ArrayList 类型非常实用，所以对集合的实现，其基本上成为了一种标准的选择。但是，选用具体的集合类型时还要考虑运行时间和存储空间等实际要求。若已经选用了 ArrayList 类型，同时又不希望重复元素的出现，则需要在加入元素时进行重复值的检查。

代码 6.3 中，实例变量 director 实际具有的类型应为 List<Employee>，这是泛型，在原始的类图中并没有体现。若要将这些模板类型也描述出来，可以采用图 6.6 所示的形式。泛型的描述方式是类的右上角带有虚线的小矩形，表示模板参数。图中的虚线带三角的箭头表示此泛型实际上是一个接口，ArrayList<Employee>对该接口进行了实现。

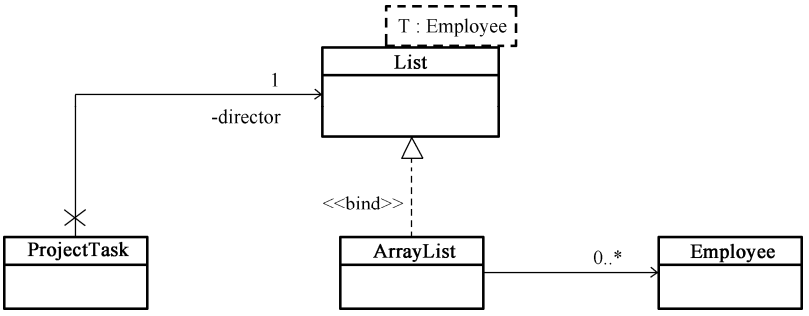


图 6.6　使用 List 模板类型的类图

6.4　对象归属关系的实现

前面的章节已经对多重性“0..1”的实现方式进行了解释，这里还涉及对象销毁后内存释放的问题，Java 具有自动的垃圾回收机制，但是对于 C++开发者和 UML 建模者来说，还要进一步深入了解对象归属的概念。在 UML 语义中，对象归属关系实际上是语义更强的关联关系。图 6.2 中的(7)和(8)两种情况都使用了菱形来表达不同的对象归属关系，下面分别进行说明。

6.4.1　聚合关系

图 6.7 描述的基本含义与图 6.3 相同，但还包含了一个深层的含义，即一个 Employee 对象可以同时参与多个 ProjectTask，也就是该 Employee 对象可以在这些 ProjectTask 中共享。这种表示对象之间归属情况的关联关系称为聚合关系，它的 Java 实现如代码 6.4 所示。

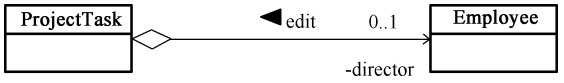


图 6.7　聚合关系的表示

代码 6.4　聚合关系的 Java 实现

```
public class ProjectTask{
    private Employee director = null;
    public Employee getDirector(){
        return director;
    }
    public void setDirector(Employee director){
        this.director = director;
    }
}
```

这段代码与代码 6.2 中两者关联关系的实现基本一样，但确实符合图 6.7 的语义描述。假设一个 ProjectTask 对象 ta 有一个对 Employee 对象的引用，则另外一个任务对象 tb 可以通过 ta.getDirector()获取并持有对这个 Employee 对象的引用。

```
tb.setDirector(ta.getDirector());
```

若使用 C++实现相同的语义，则对应有两种可能的形式。一种形式是通过常用的指针，其对应的实现如代码 6.5 所示。

代码 6.5　聚合关系的 C++实现 1

```
//ProjectTask.h
class ProjectTask{
private:
    Employee* director;
}
//ProjectTask.cpp
Employee* ProjectTask::getDirector(){
    return director;
}
void ProjectTask::setDirector(Employee* director){
    this->director = director;
}
```

另外一种形式是使用 C++中的地址操作符&，获取对象的地址用于函数参数中，其对应的实现如代码 6.6 所示。

代码 6.6　聚合关系的 C++实现 2

```
//ProjectTask.h
class ProjectTask{
private:
    Employee director;
}
//ProjectTask.cpp
Employee& ProjectTask::getDirector(){
    return director;
}
void ProjectTask::setDirector(Employee& director){
    this->director = director;
}
```

从面向对象的角度思考这些关联关系的代码实现，就会发现一些问题，面向对象对实例变量的封装性在实现上被破坏了，并没有设计得那么美好。例如，某个外部类如果获得了一个 ProjectTask 的引用，那么它可以通过公共方法 getDirector()获得私有的 Employee 对象，然后调用 Employee 的所有公共方法对其进行任意的修改。

```
ta.getDirector().setLastname("Whatever")
```

避免上述问题的一个可能做法是：使 ProjectTask 类不返回对实际对象的引用，而是只返回一个中间设置的一个接口类型。接口是一种特殊的抽象类，要求其中没有具体实现的方法。采用接口的设计方案在图 6.8 中进行了描述。

图 6.8 对 Employee 类也进行了内容上的丰富，加入了部分需要的方法和实例变量。这里假定 ProjectTask 类只使用 Employee 类中的一些读取信息的方法，通过加入一个新的接口

TaskDirector 并加入用于读取方法的声明，它们的具体实现存在于 Employee 类中。这样，ProjectTask 类不再直接使用具体的员工对象，而是间接地使用了一个实现了接口 TaskDirector 的对象。因为该接口只把需要的方法暴露给外部，从而保证了信息的安全性。当然，这种安全性是相对的，因为这样并不能阻止通过强制类型转换把该接口转换为 Employee 类的对象。

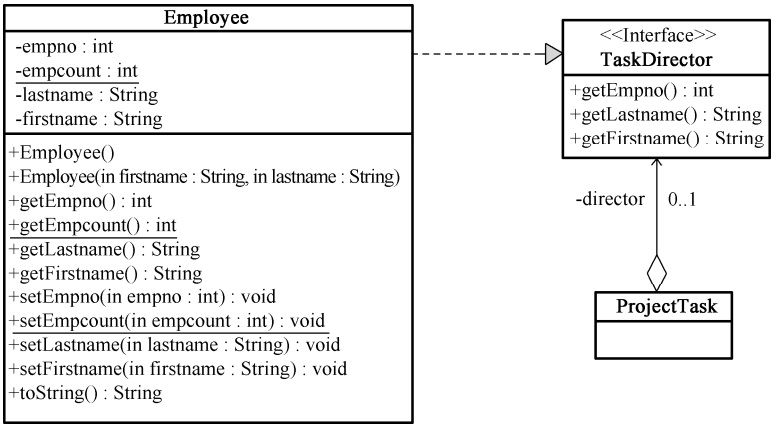


图 6.8　使用接口优化设计

另外一种不破坏封装性的设计想法是 ProjectTask 类的方法不返回任何 Employee 类的对象，这也意味着只有 ProjectTask 类允许对 Employee 对象进行操作。若需要对外提供修改 Employee 对象的服务，则需要在 ProjectTask 类中提供一个具有修改能力的公共方法，如：

```
public void updateLastNameDirector(in lastname: String)
```

其是通过调用 Employee 类中的 setLastname()方法，并通过参数的传递完成实际的修改。这种方式的在聚合关系中很常用。

图 6.9 描述的是 Employee 类对接口 TaskDirector 进行实现的一种简单形式，UML 提供了构造型<<Interface>>和用于便捷表示的“棒棒糖”图形用于接口的描述。

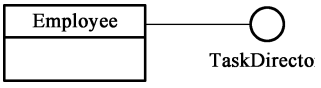


图 6.9　棒棒糖形式的接口描述

6.4.2　组合关系

除了以上介绍的聚合关系，还有一种归属关系——组合关系，图 6.10 所示的实心菱形就是这种关系的表示。组合关系的语义描述了 Employee 类和 ProjectTask 类之间的一种存在依赖性，即如果 ProjectTask 类的对象被销毁，其所属的所有 Employee 类对象(工作人员)会被同时删除，实际的含义是工作关系的解除。

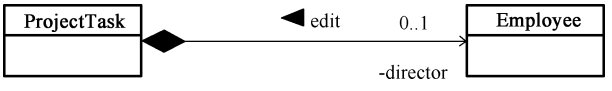


图 6.10　组合关系的表示

根据实际业务的情况，这种语义可能会有不同的实现。一种较为直接的方法是仍然使用上面的对简单聚合关系的处理方法。不过，这样做需要考虑当所属的某个对象被删除后，其他对象可能仍持有对它的引用，因此每个使用者在访问这个对象前需要判断它是否仍然存在。

更为周全的实现方法是使 ProjectTask 对象持有对所有 Employee 对象的完全代理权，即只有 ProjectTask 对象能够直接使用 Employee 对象并对其进行操作，如对一个 Employee 对象的修改只能借助以下代码中的 updateLastNameDirector() 方法实现。必要时，ProjectTask 类可另外提供 getDirector() 方法返回一个 Employee 对象的复制，而不是引用。这样，该方法的调用者可以在其本地对 Employee 对象进行任意形式的使用，而不会影响原始的 Employee 对象。代码 6.7 是上述方案的 C++实现。

代码 6.7　组合关系的 C++实现

```
//ProjectTask.h
class ProjectTask{
private:
    Employee director;
}
//ProjectTask.cpp
Employee ProjectTask::getDirector(){
    return director;
}
void ProjectTask::setDirector(Employee director){
    this->director = director;
}
```

在 get 方法的返回值和 set 方法的赋值中，每次都要确保对象副本的创建。在 Java 中，这种实现方式还没有得到直接的支持，但返回对象副本可以借助 Java 提供的克隆(Clone)机制，如代码 6.8 所示。

代码 6.8　组合关系的 Java 实现

```
public class ProjectTask{
    Employee director = null;
    public Employee getDirector(){
        return director.clone();
    }
    public void setDirector(Employee director){
        this.director = director.clone();
    }
}
```

使用 clone() 方法时需要注意，其默认的实现是对对象的浅层复制，即内部含有的对其他对象的引用地址会被复制，若需要真正的副本(深层复制)，则需要对 clone() 方法进行重写(Overwrite)。

6.4.3 依赖关系

到目前为止，我们讨论的都是类与类之间存在的静态联系，表现为关联关系存续期间的保持性。对应地，类间还存在一种语义强度较弱的依赖关系，表现为对象之间瞬时的联系，如将某些类向其他类进行传递。与关联关系不同的是，这种瞬时的联系并不在相关的对象间保持。

从语义上看，依赖关系可以用来笼统地描述两个对象之间可能的任意联系。图 6.11 是 UML 对这种依赖关系的描述，这里也可以使用<<include>>代替<<uses>>。一般来说，如果不影响类图的清晰性和阅读性，关联关系和依赖关系都可以在设计类图中说明，这种动态的瞬时关系的实现形式如代码 6.9 所示，其中，ProjectTask 对象 ta 在加入当前选定的项目对象后，并不在该 ProjectMan 对象中保持。

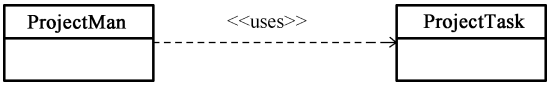


图 6.11　依赖关系的表示

代码 6.9　依赖关系的一种实现形式

```
//项目管理类
public class ProjectMan {
    private Project selectedProject;
    public void addProjectTask(String name){
        ProjectTask ta = new ProjectTask(name);
        selectedProject.addTask(ta);
    }
}
```

6.5　软件架构的实现

软件架构描述了软件系统基本的结构组织或纲要，通常提供一组事先定义好的子系统，指定它们的责任，并把它们组织在一起。对于软件架构的实现，重要的是要能够体现出其设计的思想。本部分主要讨论 MVC 结构的实现过程，并通过该过程对设计的灵活性进行解释。

MVC 的核心思想是将数据本身与其修改的方式及数据的展现形式进行分离。MVC 提供的模式，使得数据能够以各种不同的修改方式进行处理，而不影响对数据的管理和对外展现的形式。

图 6.12 是对 MVC 结构的示意性描述，图 6.13 给出了一个 MVC 的简单实现界面。为了说明上的方便，这里的模型其实只是一个简单类型的变量，为能够对该变量的值进行显示，把窗口作为一个视图并以滑块的形式显示该变量的值。另外的窗口代表一个控制器，通过两个按钮实现对模型中变量值的修改。

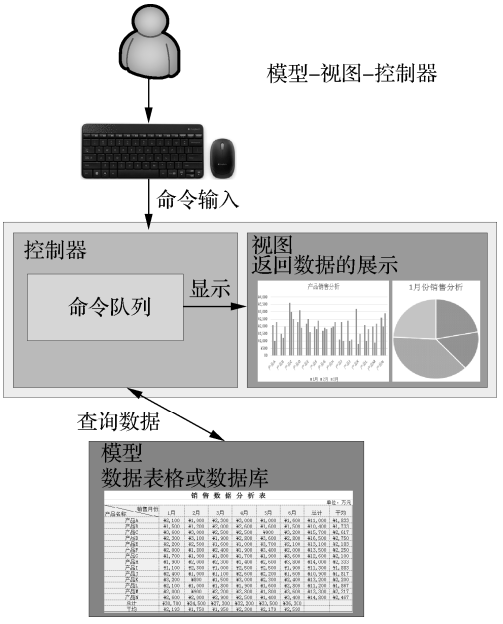


图 6.12　MVC 结构示意图

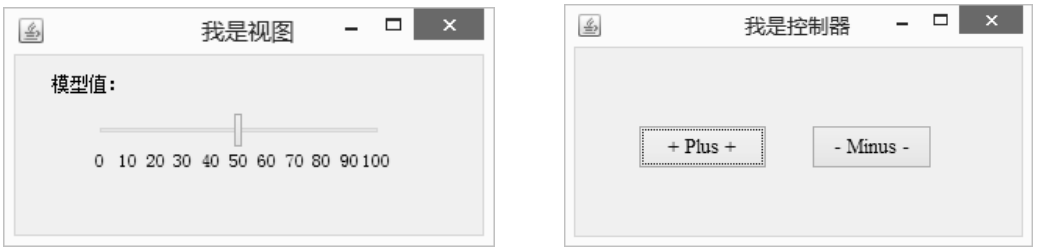


图 6.13　MVC 实现示例

MVC 的构建通过两个阶段进行。第一阶段，与 MVC 模式相关的对象被创建并进行关联；第二阶段，实现对 MVC 结构的实际使用。这个过程在图 6.14 所示的顺序图中进行了说明，第一阶段要创建模型对象，然后可以创建任意多的控制器对象和视图对象，在此过程中必须要建立它们与模型对象的联系，以便模型对象对它们进行注册和通知。该例中，首先创建了一个控制器对象，然后创建了一个视图对象，并随后在模型类处进行了注册。

对模型的修改从控制器开始，修改的来源可能多种多样，控制器对象将修改发送给模型对象，然后模型对象将修改通知给所有事先注册的视图对象。这里有两种通知视图对象的方式：一是模型对象本身将修改的值在消息中直接传递，这种方式适用于较小数据量的修改；二是只通知相关视图对象修改的状态，然后由各个视图对象在合适的时机通过调用模型对象的访问方法获取实际模型的值，这种方式适合大数据量的修改，而视图对象可能只关心其一小部分修改结果的情况。

图 6.15 给出了一个适合 MVC 结构的类图。为了使不同的视图对象能够在模型类中进行注册，以便进行通知，所有的视图对象都需要实现一个统一的接口，通过此接口能够对所有注册的视图对象进行统一管理。模型类实现如代码 6.10 所示。

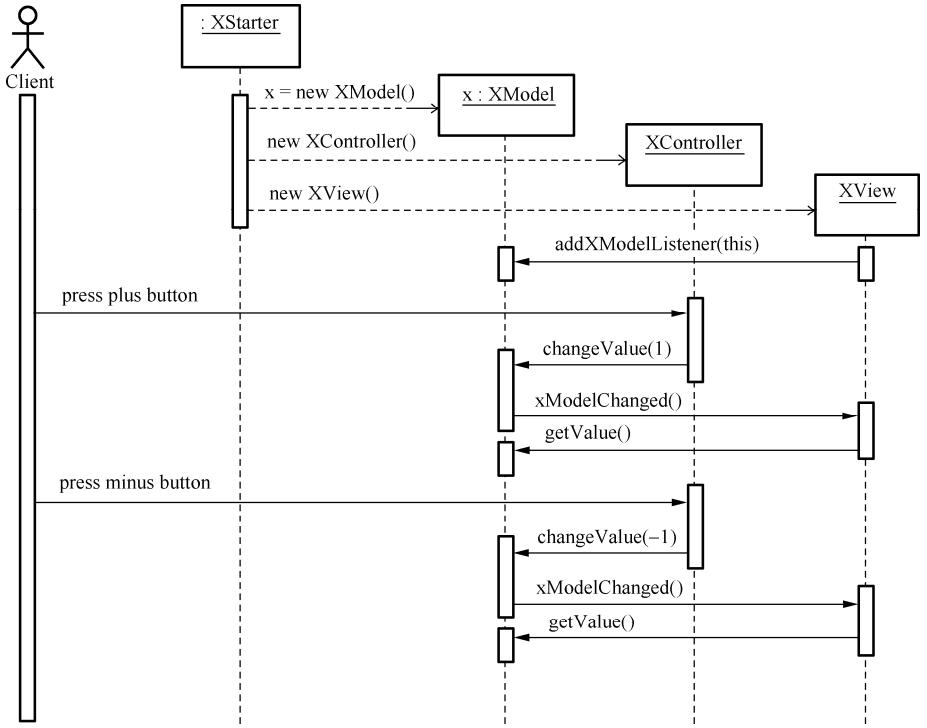


图 6.14　MVC 的构建过程

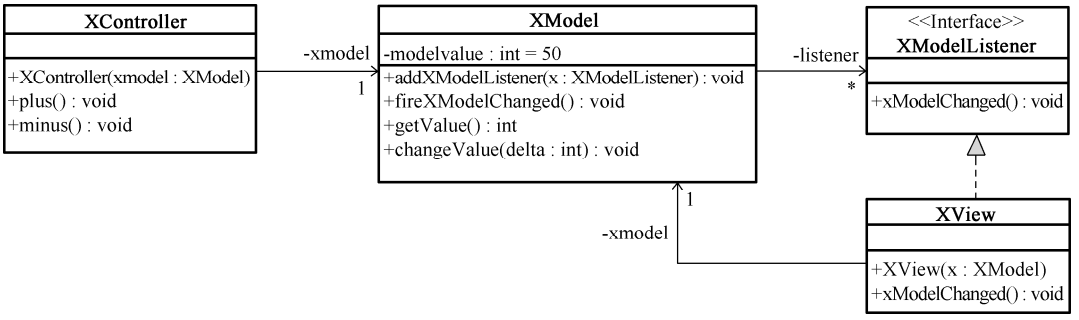


图 6.15　MVC 的类图

代码 6.10　模型类的实现

```
import java.util.*;
public class XModel{
   private ArrayList<XModelListener>listener =
                new ArrayList<XModelListener> ();
   private int modelvalue = 50;
   //Management of Model Listeners.
   public void addXModelListener(XModelListener x){
      listener.add(x);
   }
   //Notification to Listener
```

```
    private void fireXModelChanged(){
        for(XModelListener x: listener)
            x.xModelChanged();
    }
    //Read from Model
    public int getValue(){
        return modelvalue;
    }
    //Modification of Model
    public void changeValue(int delta){
        modelvalue += delta;
        fireXModelChanged();
    }
}
```

模型类含有的实例变量由两部分构成：第一部分是实现对已注册视图的管理，这里是通过一个 listener 类型的数组链表进行的；第二部分是实际的模型值，这里只使用了一个数值类型的变量进行示意性的说明。对模型中含有的值来说，通常至少要提供其访问的方法，即 get 方法。还包括一个在模型中进行注册的方法 addXModelListener()。如果要对模型值进行修改，可以在模型类中为控制器实现不同的修改方法，此例中提供了一个唯一的修改方法 changeValue()。如果模型值被修改了，就需要通知所有的注册视图，这是通过调用本地方法 fireXModelChanged()实现的。为了使模型和视图顺利配合，每个视图类必须实现接口 XModelListener，如代码 6.11 所示，视图类的实现如代码 6.12 所示。

代码 6.11　模型接口的实现

```
public interface XModelListener {
    public void xModelChanged();
}
```

代码 6.12　视图类的实现

```
import javax.swing.*;
public class XView extends JFrame implements XModelListener{
    private XModel xmodel;
    private JLabel jlabel = new JLabel ("模型值：");
    private JSlider jslider = new JSlider(JSlider.HORIZONTAL,0,100,50);
    public XView(XModel x){
        super("我是视图");
        xmodel = x;
        xmodel.addXModelListener(this);
        //Rest for display
        getContentPane().add(jlabel);
        getContentPane().add(jslider);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
```

```
            setSize(250,60);
            setLocation(0,0);
            setVisible(true);
        }
        pub1ic void xMode1Changed() {
            jslider.setValue(xmodel.getValue());
        }
    }
```

视图的实现主要借助的是代码 6.12 中的阴影标注，其余代码是 Java 图形包 Swing 中的一些辅助代码。每个视图对象需要知道模型对象的存在，并建立起与之对应的联系，本例中，这些信息是通过实例变量 xmodel 保存的，并在视图类的构造函数中初始化。为了实现对修改的通知，接下来的步骤将视图在模型类中进行了注册。视图类必须实现 XModelListener 接口。如果视图对象接到模型类修改的通知，它可以通过模型类提供的访问方法 getValue() 获取修改后的模型值。

控制器类的实现如代码 6.13 所示。每个控制器对象必须能够关联到相关的模型类上，本例中这些信息是通过实例变量 xmodel 保存的，并在控制器类的构造函数中初始化。若控制器对象需要修改模型值，则通过模型类提供的修改方法进行。

代码 6.13　控制器类的实现

```
import java.awt.FlowLayout;
import java.awt.event.*;
import javax.swing.*;
public class XController extends JFrame{
    private XModel xmodel;
    public XController(XModel x){
        super("我是控制器");
        xmodel = x;
        getContentPane().setLayout(new FlowLayout());
        JButton plus = new JButton("+ Plus +");
        getContentPane().add(plus);
        plus.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                xmodel.changeValue(1);
        }});
        JButton minus = new JButton ("- Minus -");
        getContentPane().add(minus);
        minus.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                xmodel.changeValue(-1);
        }});
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(250.60);
        setLocation(0,90);
```

```
            setVisible(true);
        }
    }
```

MVC 结构本身提供了一般性的解决方案，但本质上并不强求实际的解决方案是完全一样的。在应用中，可以根据需要进行调整，加入一些必要的设计变化，从而使方案更加适合业务场景的需要。例如，如果能确定只有一个视图对象在模型类中注册，那么可以不引入接口 XModelListener；但如果仅存在一个视图，使用 MVC 的架构思想来组织也是好的。在这种情况下，模型中的变量 listener 不再是一个集合，而是一个简单的对视图对象的引用。MVC 对各部分的划分，有助于理清各部分的职责，从而为系统带来一个清晰的设计结构。另外，对视图类的管理也可以不在模型类中进行，可以使用控制器类来负责所有对视图类的管理和模型值的修改通知。

6.6 其他物理实现

在实际的代码实现中，必须考虑到非功能性需求带来的影响，因为非功能性需求在某种程度上对实现方式的选择会带来决定性的作用。UML 类图主要用来对功能性需求进行建模，对非功能性需求的描述能力往往力不从心。

另外，将所有的项目信息都并入同一模型中进行展示也是不明智的，因为这将导致模型本身的烦琐且难以阅读，软件系统的各种信息应以不同的视图进行重点展示，这样才更有意义。视图的概念在第 4 章已经介绍了，它用来表示对未来的软件进行观察和理解的角度，每种角度专注于系统某方面的特征，最终的目的是未来的软件能够满足每个视角的要求，并通过对它们的集成，保证最终实现的系统是全面而且成功的。软件架构的“4+1”视图模型是对上述思想的归纳：逻辑视图应主要包括系统的功能性需求及类模型；开发视图应包括对子系统和接口的描述；进程视图应包括处理流程、并行性及同步等策略描述；物理视图包括对目标硬件及网络等的描述。UML 中的有些图形可以用在多个不同的视图中，但在每个视图中强调的是模型中不同的侧面。作为补充，这里着重说明 UML 在进程视图和物理视图中提供的表示模型和实现形式。

图 6.16 描述了一个可能的、独立运行的进程或者线程及其关联的活动对象。在 Java 中，它们是一些从 Thread 类继承或在 Runnable 接口实现的类，具有 run()方法。它们在 UML 中另外的描述形式是使用构造型，即用一个标有<<thread>>或者<<process>>的活动类进行表示。

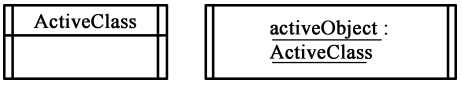


图 6.16　活动类及其对象

类和其对应的包开发完成后，需要将完成的系统划分成可以运行的单元，并进行安装。一般来说，这些单元打包成一个或者多个软件包，称为构件。在 Java 中，一般将其打包成*.jar 的形式。

图 6.17 给出了对构件进行描述的构件图，它具有两种描述方式。构件图能够描述多个构件的构成及它们之间的联系。在两种描述中，除了构件的名字，还提供了构件的规格说明和需要的接口等信息。构件提供的服务常使用“棒棒糖”的图形方式进行描述，并在其圆形处给出接口名。构件需要的接口通过一个在端点处的半圆形的符号表示，可以形象地理解为一个可以与其他构件的接口进行连接的插座，并与这些构件一同工作。

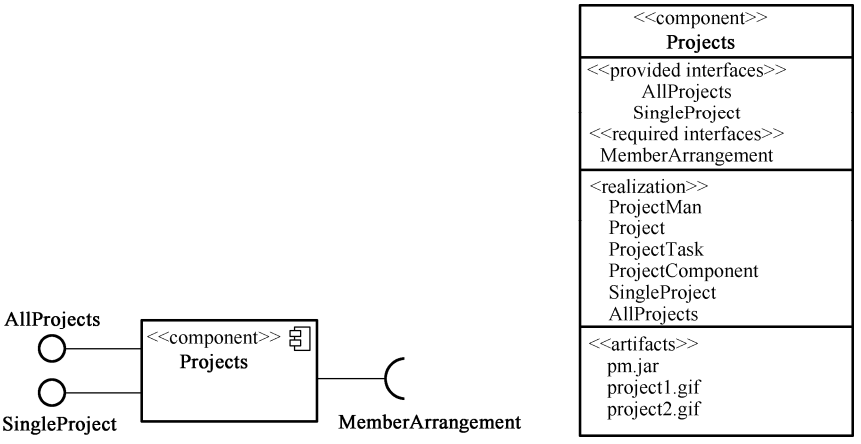


图 6.17　构件图

在每个构件的详细描述中，还可以通过<<realization>>进一步说明该构件中包含的类。此外，还可通过<<artifact>>描述与该构件相关的其他制品。构件中除了程序包，还包括其他数据及图片等内容。

图 6.18 给出了 UML 中部署图的一个例子，图中描述了软件构件最终在物理机器上部署运行的情况。除计算机信息外，还可以具体指定它们之间连接的类型及多重性。计算机作为硬件节点，使用了长方体进行描述，在其内部安装部署可以运行的程序(<<executable>>)和其他相关的制品(<<artifacts>>)。对于网络连接，还可具体指定网络的类型和带宽。

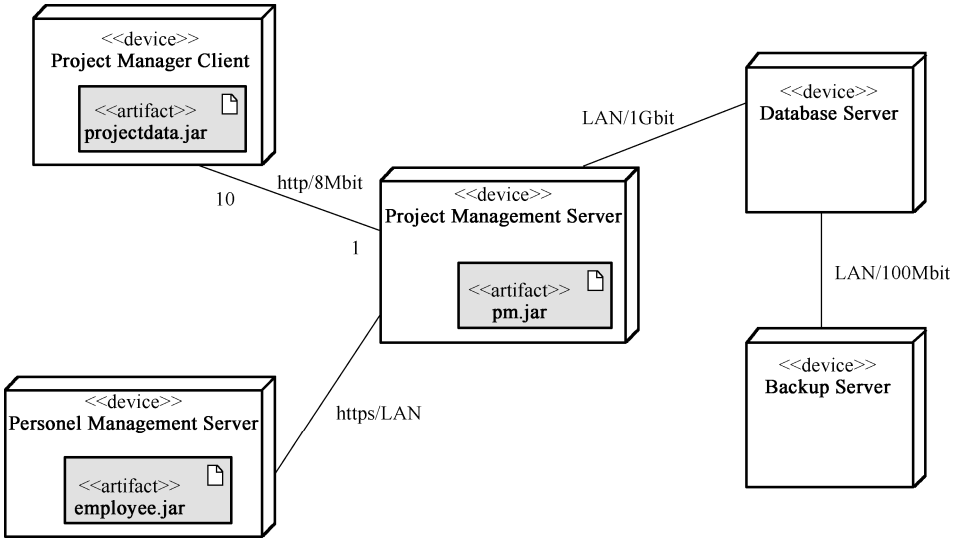


图 6.18　部署图

6.7 习题

1．详细说明图 6.19 所示的类图中两个关联关系表示的含义：

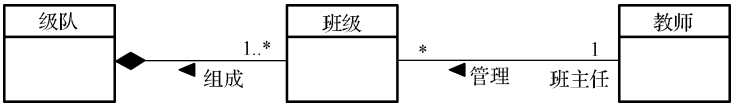


图 6.19 类图

2．下面给出的代码 6.14 是使用 Java 语言对 4 个类的实现。为了简短起见，一些类定义的细节没有给出，另外还省略了相应的 set 和 get 方法。对这些代码使用逆向工程，要求使用 UML 给出它们的设计类图。

代码 6.14　类的实现

```
class TestQuestion {
    public static void main(String [] args){
        CardList myCardList = new CardList();
        Card newCard;
        for (int i = 1; i<= 2; i++){
            int j = i + 5;
            newCard = new Card(i, j);
            myCardList.insertCard(newCard);
        }
    }
}
class Card {
    public Card(int s, int r) { suit = s; rank = r; }
    protected int suit;
    protected int rank;
    //...
}
classListElement {
    protected ListElement prev;
    protected ListElement next;
    protected Card card;
    //...
}
class CardList {
    // This method creates a new list element on the heap associated with
    // the input card and inserts into the beginning of this list.
    public void insertCard(Card card) { ... }
    // This method removes the input card from this list.
```

```
    public Card removeCard(Card card) { ... }
    // This method returns the first card of this list.
    public Card getFirst() { ... }
    //...
    protected int noOfElements;
    protected ListElement first;
    protected ListElement last;
}
```

3．某位系统分析人员针对高校学生选课系统的需求设计了分析类图的草图，如图 6.20 所示。

(1)教务人员通过职工号和密码登录系统，录入课程信息。课程信息包括：课程编号、课程名称、授课时间、授课地点。并为每名教师排课：一名教师可以教授 0 门或多门课程，1 门课程也可以由 1 或多名教师教授。

(2)学生通过学号和密码登录系统，选择课程。系统要求 1 门课程至少要有 10 名学生选课才能开课，1 名学生至少要选择 1 门课程。

(3)教师通过职工号和密码登录系统，获取课程的学生名单。

(4)课程结束后，教师通过该系统录入学生成绩。

(5)学生可通过该系统查询自己的课程成绩。

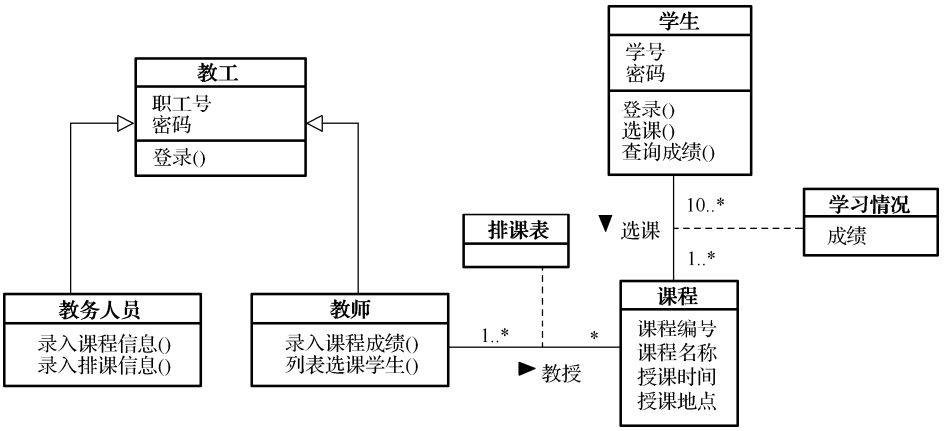


图 6.20　类图草图

请基于图 6.20 进行细化和完善，尽可能补充需要的信息，完成该系统的设计类图，并尝试使用 Java 或 C++语言编写该系统的代码框架。

第 7 章　类的详细设计

类的详细设计以概要设计说明书为基础，完成各模块的算法设计、用户界面设计及数据结构设计的细化等。在系统的需求分析和概要设计中，通过对业务类图的构建给出了进行初始实现的类结构的建议。第 6 章对从设计类图到可运行程序的代码框架做了说明。对于开发者来说，详细设计的主要工作就是在每个类的方法中补充对应的业务实现。这也是最令人兴奋和最具挑战性的任务，因为这是对开发者创新和专业能力的考验，真正将设计中非形式化的描述准确无误地转换成期待的、能够运行的程序功能。

本章首先对传统过程化的详细设计进行说明，这适用于类中方法的设计。然后对状态图进行介绍，其提供了对对象生命周期的描述。状态图能够说明每个对象在什么环境中、在何种事件的作用下如何响应，并提供具体类的行为描述。最后，对对象约束语言进行介绍，这种机制可以用来在模型中形式化地约束具体的业务或边界条件。

7.1　详细设计的主要活动

详细设计应忠实概要设计的结果，不得破坏概要设计方案；对每个模块应给出详细设计的方案说明，并最终形成详细设计说明书，包括详细的模块实现方式、实现模块功能的类及具体方法的流程等，一般包括以下活动内容。

(1) 为每个模块进行详细的算法设计。用图形、表格、语言等工具将每个模块处理过程的详细算法描述出来。

(2) 为模块内的数据结构进行设计。对于需求分析、概要设计确定的概念性的数据类型进行确切的定义。

(3) 为数据结构进行物理设计。确定数据库的物理结构，物理结构主要指数据库的存储记录格式、存储记录安排和存储方法，这些都依赖于具体使用的数据库系统。

(4) 其他设计。根据软件系统的类型，还可能进行输入/输出格式设计、人机交互设计。对于一个实时系统，用户与计算机频繁交互，因此还要进行交互方式、内容、格式的具体设计等。

(5) 编写详细设计说明书。

(6) 评审。对处理过程的算法和数据库的物理结构进行评审。

在传统的开发模式中，常用于描述详细设计的工具包括图形、表格和语言等，这些方法在面向对象领域也使用，尤其适用于类中方法实现流程的描述。

为了使非形式化描述的类图和代码实现之间的差距不会过大，而且不会引入过多的假设和错误，UML 提供了在类图中对实现细节进行准确定义的机制。利用这种机制，可以在图形中对具体的业务约束或边界条件进行说明，通过使用独立的对象约束语言(Object Constraint Language，OCL)表达类与类之间更复杂的业务关系和约束，并且使用 OCL 可以准确定义出某方法调用的预期结果，不需要给出具体的实现内容。

7.2 类方法的详细设计

传统的函数、过程、子过程等模块并不像面向对象那样封装数据，它们是面向功能的行为实现。20 世纪 70 年代，人们提出了“结构化程序”设计的理念，仅使用几种不同类型的程序块就可以构造任何程序结构，每种类型的程序块对应不同的逻辑结构，分别是顺序结构、选择结构和循环结构。顺序结构是任何程序逻辑的基础，表示动作按照指定的顺序依次执行。选择结构指定两个或若干个条件，构成不同的逻辑分支，实际执行的流程只能是满足条件某个的分支。当循环条件得到满足时，循环结构能够使得某段程序反复运行。这 3 种逻辑结构是结构化程序设计的基础。除此之外，结构化程序设计还要求每个逻辑程序块只有一个入口和一个出口。图 7.1 给出了两个非结构化程序设计的例子。

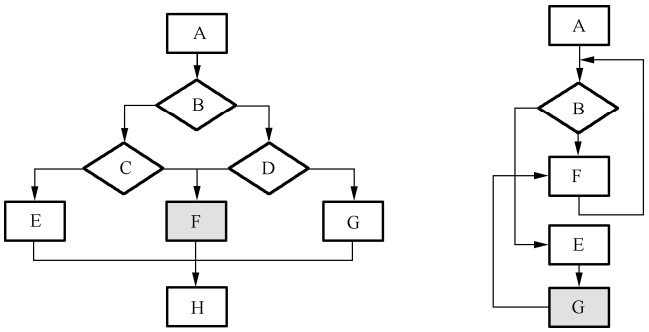


图 7.1 非结构化的程序设计

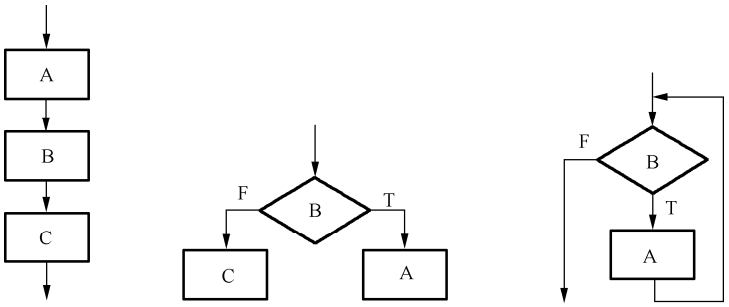
引入结构化程序设计的目的是简化软件设计的过程，仅使用有限的可预测的操作完成相应的算法流程。通过复杂程度的度量，可以看到结构化程序设计能提高程序的可读性、可测试性及可维护性。这几个基本的逻辑结构是人们理解业务逻辑最简单的模式[①]，通过模式，为开发者提供统一的理解和感悟，从而提高可读性、可理解性。

7.2.1 图形工具

1. 程序流程图

程序流程图(Flowchart)是最常用的一种描述详细设计的工具。它的广泛使用源于其简单、直观、易于学习。矩形代表一个处理步骤，菱形表示逻辑选择，箭头所指表示控制流的方向。图 7.2 给出了三种基本结构的程序流程图表示方法。顺序结构中的两个处理步骤通过一条有向控制直线连接。选择结构中有两个分支，描述了 if-then-else 逻辑，当条件为真或者假时，对应的右分支或左分支被执行。当然，经常也会遇到多分支选择的情况，可以由多个二分支选择结构扩展构成，很多程序设计语言中也提供了简单、特有的 switch-case 结构。循环结构有两种不同的描述方式：一种是 While 循环，一种是 Until 循环，它们的主要区别是循环条件结构和逻辑响应方式的不同。

① 模式即知识，是经验的总结，模式的概念在后续章节中还会专门介绍。



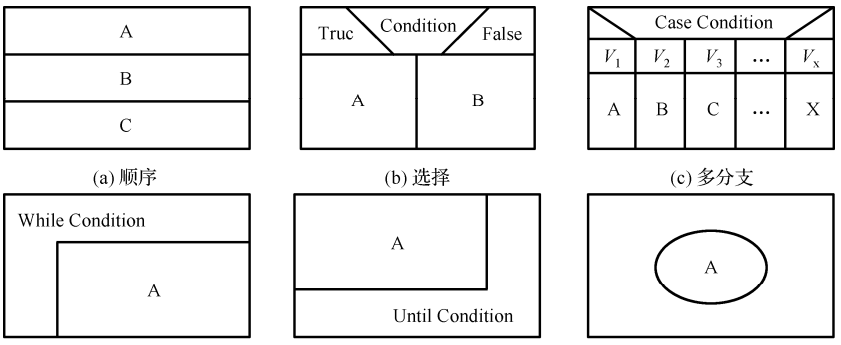
(a) 顺序　　(b) 选择　　(c) 循环

图 7.2　三种基本结构

一般情况下，采用结构化程序设计会使设计更易读、更直观，但程序流程图的最大问题是表示控制流程的箭头可以比较随意地绘制，使得设计人员可能在不经意间违背了结构化程序设计风格，造成设计出的程序流程混乱、难懂。因此，使用程序流程图进行详细设计时，应尽量保持结构化的特点，限制控制流的随意跳转。但是，在一些追求高效率、实时性、代码精简度的情况下，会适当使用 goto、break、continue 等容易违背结构化设计的语句。

2．盒图

盒图又称 N-S 图，由 Nassi 和 Schneiderman 提出，是一种符合结构化程序设计原则的图形描述工具。盒图中没有表示控制流程的箭头，因此不允许随意转移控制。它的基本逻辑块如图 7.3 所示，在每个逻辑块中，语句被限制在一个封闭的“盒子”中，各种逻辑盒子组合嵌套，构成更大的盒子。程序流程的结构在盒子中隐式体现。



(a) 顺序　(b) 选择　(c) 多分支

(d) While循环　(e) Until循环　(f) 过程调用

图 7.3　盒图

使用盒图作为详细设计的工具，可使开发者逐步养成用结构化的方式思考问题和解决问题的习惯。盒图的主要问题是绘制起来不是很方便，尤其是当问题逻辑比较复杂时，绘制盒图可能会很烦琐。满足结构化设计的程序流程图都能够转换为对应的盒图，否则不行，因此可以使用盒图来验证设计是否符合结构化的要求。

3．PAD 图

PAD（Problem Analysis Diagram）图自 1973 年由日本日立公司发明以后，已得到一定程度的推广并受到国际标准化组织的认可。PAD 图从程序流程图演化而来，使用结构化程序设计

的思想表示程序设计的逻辑结构，绘制起来比较方便。PAD 图使用二维树形结构表示程序的控制流，将这种图翻译成程序代码也比较容易，如图 7.4 所示。

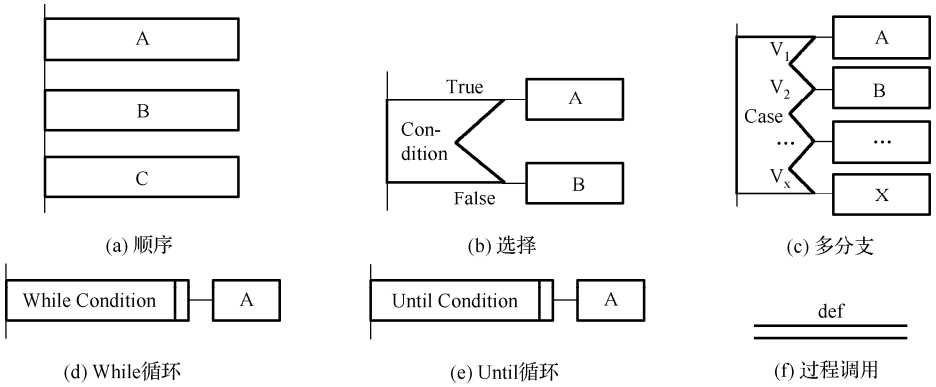


图 7.4　PAD 图

PAD 图描述的程序的层次关系表现在纵线上，每条纵线表示一个层次，PAD 层次关系从左至右逐层展开，向右延展。控制流程从最左主线的上端开始，自上而下依次执行，每遇到判断或循环，就进入下一层，然后从表示下一层的上端开始执行，直到下端，再返回上一层的转入处。如此继续，直到主线的所有内容都执行完成。

7.2.2　表格工具

判定表是一种进行详细设计的表格工具，又称决策表[①]。判定表适用于描述判断条件较多、各条件又相互组合、有多种决策方案的情况。判定表有着准确而简洁的描述方式，能够将复杂的条件组合与对应的执行动作相对应，如表 7.1 所示。

表 7.1　一个判定表的例子

规则#	1	2	3	4	5	6	7	8	9	10	11	12
机器功率 w(马力)	A	A	A	A	B	B	B	B	C	C	C	C
运行时长是否大于 10 年	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
是否有维修记录	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y
送外维修	※	※	※	※	※	※	—	—	※	※	—	—
本厂维修	—	—	—	—	—	—	※	—	—	—	※	※
本车间维修	—	—	—	—	—	—	—	※	—	—	—	—
$A: w>50, B: w<20, C: 20\leqslant w\leqslant 50$												

这个例子描述的是某工厂机器维修的方式：对功率大于 50 马力(1 马力=735.49875W)的机器或已运行 10 年以上的机器，应送到专业的维修公司处理；若功率小于 20 马力，并且有维修记录，则在车间维修；否则送到本厂的维修中心维修。

判定表由 4 个部分构成，分别是条件列表、条件组合、动作列表及动作入口。每个条件对应一个变量、关系或者预测，如上例中的机器功率、运行时长、维修记录；条件组合是各种条件可能取值的所有组合，若每个条件有真、假两种取值，则 n 个条件的取值组合数量为 2^n 个。在表 7.1 中，由于功率有 3 种取值，所以理论上的条件组合数为 $3\times2\times2=12$；动作是指

① 判定表除了可以用来进行详细设计，在软件测试中也有应用。

要执行的过程或操作列表，如表 7.1 中的送外维修或者送本厂维修；动作入口是指某个条件组合下与动作的对应，与条件组合一起构成判定表的一列，也称规则。

判定表初步生成后，可以尝试将具有相同动作入口的条件组合进行合并化简，即找出对动作结果没有影响的条件(真假都包含)，我们使用“—”来表示对此条件的不关心或不适用，如表 7.1 中规则 11 和规则 12，对应的动作都为“本厂维修”，但只有一个条件不相同且一个取真值，另一个取假值，表明这两条规则对应的结果与是否有维修记录是不相关的，因此两者可以合并为表 7.2 中的规则 4。

表 7.2　判定表的化简

规则#	1	2	3	4	5
机器功率 w(马力)	A	—	B	C	B
运行时长是否大于 10 年	N	Y	N	N	N
是否有维修记录	—	—	N	—	Y
送外维修	※	※	—	—	—
本厂维修	—	—	※	※	—
本车间维修	—	—	—	—	※
$A: w>50, B: w<20, C: 20\leqslant w\leqslant 50$					

在化简的过程中，还需要注意任意两个条件组合之间不能有交集，也就是说，任意给定一个实例(某一台机器)，在表中只能有唯一一列规则与之匹配。在这个例子中，还有其他的化简可能，请读者自行给出。

判定表虽然能清晰地表示复杂的条件组合与应做的动作之间的对应关系，但其含义却不是一目了然的，理解它也需要有一个学习过程。而且，当数据元素的值多于两个时(如例子中的功率)，判定表的简洁程度也将下降。判定树是判定表的变种，也能清晰地表示复杂的条件组合与应做的动作之间的对应关系。

判定树又称决策树，是应用于数据分类的一种树结构。树中的每个内部节点(Internal Node)代表对某个属性的一次测试，每条边代表一个测试结果，叶子节点(Leaf Node)代表某个类别(Class)或者类别的分布(Class Distribution)，最上面的节点是根节点。判定树提供了一种展示在什么条件下会得到什么值(这类规则)的方法。图 7.5 是表 7.2 对应的一棵判定树，从中可以看到其基本的组成部分。

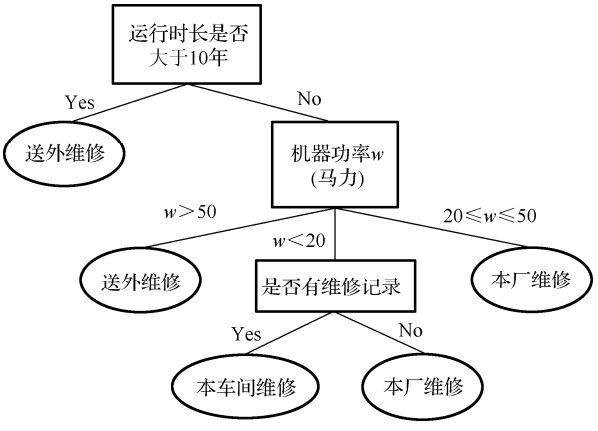


图 7.5　判定树示例

给定问题域及数据，寻找最佳判定树是 NP 困难问题，工程上常采用一些启发式的算法，经典的算法有 ID3 算法等，即决策树归纳(Induction of Decision Tree)。早期的 ID 3 算法只能就两类数据进行挖掘(如正类和反类)，经过改进后的 ID3 算法可以挖掘多类数据。

7.2.3 语言工具

程序设计语言(Programming Design Language, PDL)是一种用来进行详细设计的语言工具，又称结构化语言或伪代码。与自然语言相比，PDL 借用某种编程语言的语法构筑其逻辑流程；与编程语言相比，其中又使用了自然语言的词汇(如英语)。PDL 不能被编译或解释运行，主要供开发者使用。除此之外，PDL 还具有以下特点：

(1) PDL 采用关键字的固定语法，具有结构化控制结构、数据说明和模块化的特点；

(2) PDL 程序中会有一些能够标明程序结构的关键字；

(3) PDL 仅有少量的简单语法规则，大量使用人们习惯的自然语言；

(4) 使用 PDL 时，常常按逐步细化的方式写出程序；

(5) PDL 程序的注释行对语句进行解释，起到提高可读性的作用。

代码 7.1 是一段使用 PDL 设计的示例。

代码 7.1　伪代码示例

```
procedure: sort
    do while records remain
        read record;
        if record field 1 = 0 then
            process record;
            store in buffer;
            increment counter;
        else if record field 2 = 0 then
                reset counter;
            else process record;
                store in file;
            end if
        end if
    end do
end
```

PDL 最大的优势是容易被翻译成某种编程语言描述的代码，如 C、Java 等。同时，PDL 不是编程语言，因此不必过多担心存在语法错误，可以将精力集中在设计上。PDL 作为设计工具，没有图形工具形象、直观，但其表现形式是最接近代码实现的。

7.3 类的行为设计

类是面向对象分析和设计的基本单元，尤其是业务相关的类，其构成与需求密不可分。本节主要介绍通过类的行为和状态变化，对类内部的构成进行详细设计。

第一种类型的类，类中存在某些部分，其主要功能是对业务信息进行管理，这些业务信息包括实例变量的存储、实例变量的访问方法，它们可以进行简单的业务逻辑处理并返回具体值。这些方法的返回值一般对现有的实例变量在取值上没有任何影响，通常把这些类的实例对象称为无记忆对象(Memoryless)或无状态对象(Stateless)。

第二种类型的类，其行为大多依赖于对象当前的状态，例如，一个进行远程连接的对象，在需要连接时通常只需要对该对象进行一次通知，只有已经建立了连接，该对象才可以通过调用数据传输方法进行数据交换，最后只能对已有的连接进行关闭。在这样的情况下，要确定何时应该调用哪些方法，这依赖于该对象的内部状态。通常把这类对象称为有记忆的对象或者有状态的对象。

对象的状态及状态变化可以借助状态图(State Diagram)或有穷状态机(Finite State Machine)进行描述。对于那些具有依赖类状态的行为，借助状态图能够对其进行清晰的描述；相反，对于那些只含有对实例变量的简单操作，而且并不依赖于实例变量的行为，使用状态图对其进行描述则没有太大的意义，因为这样的类实际上只具有一个中心的状态，在这个状态下可对所有方法进行调用，而对其本身的状态不产生任何影响。

7.3.1 状态图的结构

图 7.6 给出了状态图的基本结构，图中左侧描述的是状态的一般构成模板，右侧是一个具体的状态。每个状态图中应含有一个唯一的开始状态，用一个实心圆表示，可以有多个结束状态，用半实心圆表示，开始状态和结束状态统称为伪状态。一般来说，状态图中的状态变化都是针对确定性行为的描述。也就是说，在状态一定、事件相同的情况下，该对象的下一个状态也是相同的。另外，如果图中只有一个结束状态，在较复杂的状态图中可能会有多条边交织指向同一个结束节点，为了避免这种显示上的混乱，可以使用多个结束状态，以保持图形的整洁。

状态通过状态转换进行过渡(Transition)，状态转换使用一条带有简单箭头的实线表示，连接源状态和目标状态。每个状态描述中除了状态名，还可以包含以下三个预定义事件的描述。

(1) entry：给出刚进入该状态时应该进行的动作(Action)。这里，可以是一个简单的赋值操作，也可以是对一个或多个方法的调用。

(2) do：给出在保持该状态的过程中，对象应执行的活动。这里一般对那些受时间控制行为的对象比较适用，因为它们通常要求能够持续地读取信息。

(3) exit：给出当离开该状态时应进行的动作。

这三部分描述的内容是可选的，可根据需要进行取舍。对状态转换的描述也由三部分内容构成，如图 7.6 中的状态转换所示，同样，它们也可根据需要决定取舍。这三部分的详细解释如下。

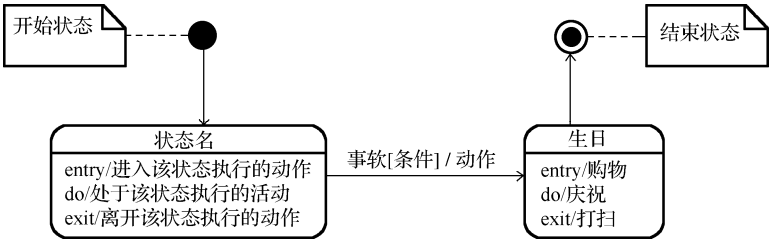


图 7.6　状态图的基本结构

(1)事件：它是转换说明的主要内容，因为状态图主要是对被动系统的描述，即对外界的激励事件进行相应的响应。一个事件可以对对象的一个方法进行调用，也可以是一个内部状态的改变，如更改变量的最终值。重要的是，要在状态图的说明或者文档中清晰地描述出那些事件。

(2)条件：状态间的转换只有在事件被触发并且满足某个特定条件时才会进行。这个条件通常返回一个布尔类型的结果并且在状态图中使用方括号表示。当然，这个附加的条件是可选的内容，当没有条件存在时，表示返回的一直是真值，意味着只要事件发生，状态间的转换就会进行。另外，布尔条件的指定通常与对象中的实例变量相关。

(3)动作：表示当转换发生时执行的一个动作，该动作执行的时机是在转换对应的目标状态的 entry 事件被执行之前，即还未进入目标状态前。

图 7.7 是一个项目类的状态图，它的具体解释是：当一个项目对象被创建时，会进入“项目创建”状态并执行 entry 部分的项目初始化，如执行一些项目基本信息的 set 方法，然后状态发生自动转移，将状态过渡到“项目计划”状态，自动转移发生在没有任何触发事件和转移条件说明的情况下。在此状态下，存在三个具体的事件可能被触发，进行不同的转换。具体来说，如果事件“新建子项目”被触发，该对象将会进入“子项目添加”状态，并加入其所包含的子项目。接下来的状态要根据方括号中指定的条件来决定哪个转换会被执行，这个条件用来判断该子项目是否具有前置项目。在活动图中使用一个小菱形描述可选的分支，状态图对于分支条件也可使用同样的描述方法，并且多个分支最后在菱形处汇集，但这里将菱形进行了省略，只是通过条件进行分支的表示，是一种简化的表示方法。通过该状态图的其他描述，还可以了解到，当该项目计划结束后，项目将由“项目计划”状态转移到“项目执行”状态，并且在这个状态下只能够进行项目工作量和完成度的更新。

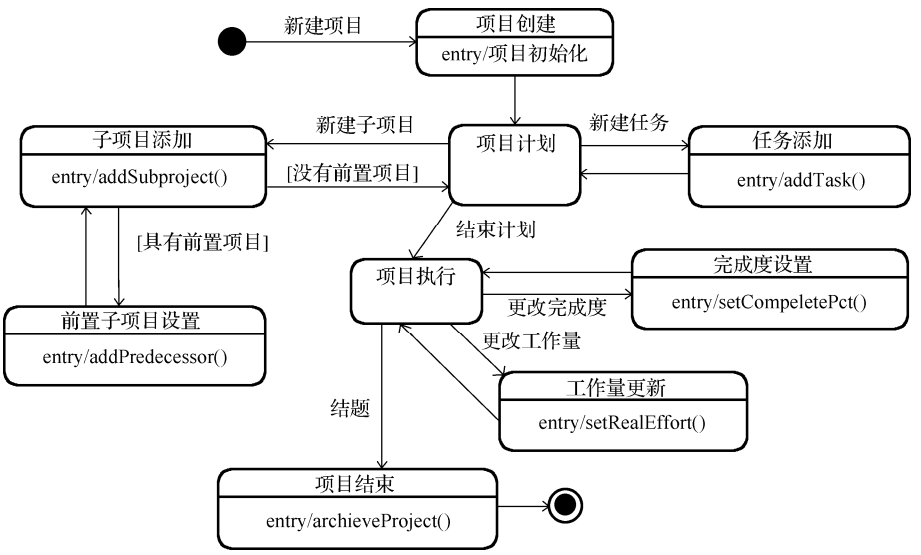


图 7.7　项目类的状态图

进一步地，需要通过文档的方式明确具体事件被触发的时机，这对于状态图对应的代码实现是很必要的。通过正式描述的状态图，可以直接翻译成运行的程序。一种直接的翻译方法是将状态图中的状态定义为若干个枚举类型值，在类中使用一个实例变量，用以记录当前对象

所处的状态。如果对象的状态可以较容易地用某单个实例变量的取值进行标识，也可以将表示状态的枚举类型和状态变量省略。

7.3.2 状态图的扩展

状态图的可视化和直观性，使其非常适用于开发小组的讨论。但如果状态图本身过于庞大，就容易让人陷入局部而丢失整体上的可理解性。因此，UML 的状态图在基本状态图的表示方法上进行了扩展，加入了一些诸如复合状态或并行状态的描述机制，但 UML 能够保证每个扩展只是使用了现有的描述机制，并对简单状态图进行的一种重新整理。下面给出几种常用的状态图扩展方式。

一种常见的情况是状态图中若干状态在同一事件作用下具有相同的行为，如对于异常的处理或者运行的停止，这样的行为可考虑进行提取。状态图可以以一种层次化的方法进行组织，每个状态通过多个子状态细化，该状态称为复合状态。在图 7.7 的基础上，图 7.8 给出了一种改进的表示，其中“活动项目”状态是一个复合状态，它通过一个独立的带有起始状态的子状态图进行了细化，并且通过事件“结题”结束“活动项目”状态。这里，外层状态“活动项目”有一个新的转换，其始于该状态边界，指向其结束状态，转换事件为“项目取消”，表示该复合状态中的每个子状态都可以在该事件的作用下转移到最终的结束状态。这样设计的好处是：对于复合状态中的所有子状态，只需要一个转换来描述它们共同的行为，节省了转换的个数，同时使图形的绘制保持整洁，不会过于杂乱无章。

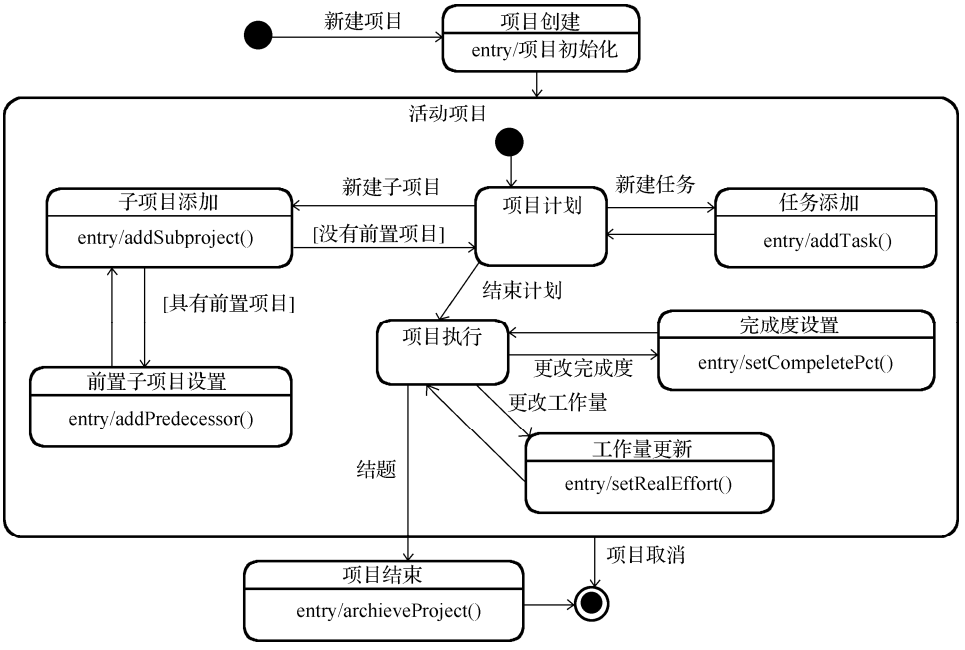


图 7.8　具有复合状态的状态图

另外，对象的状态也可能是由多个互不依赖的子状态构成的。以电子表对象为例，其对时间的显示(小时和分)有 12 小时制和 24 小时制两种方式，同时还可以实现日期和秒的显示切换，这两个行为变化序列其实是互不相关的。每种可能的行为变化，可使用一个单独的状态图

分别进行描述。一个状态图在 12 小时制和 24 小时制之间进行切换，通过一个“up”按钮触发；另外一个状态图在显示日期和秒之间切换，通过一个“down”按钮触发。

若要将两个独立行为描述的状态变化放在同一个状态图中，则组合起来就需要 4 个状态对该对象的行为进行描述，它们是“12 小时制/日期”“24 小时制/日期”“12 小时制/秒”“24 小时制/秒”。也就是说，如果每个状态图中有 m 个状态，共有 n 个状态图，若要放在一个状态图中进行描述，则总共需要 m^n 个状态表示。

图 7.9 左侧的表示方法中没有并行的组合描述，而右侧的图是一个对左侧的简化，其中对组合状态使用了多个并行的子状态图分别对两个行为变化序列进行描述。每个子状态图相互独立，组合在一起又构成一个全局的状态。

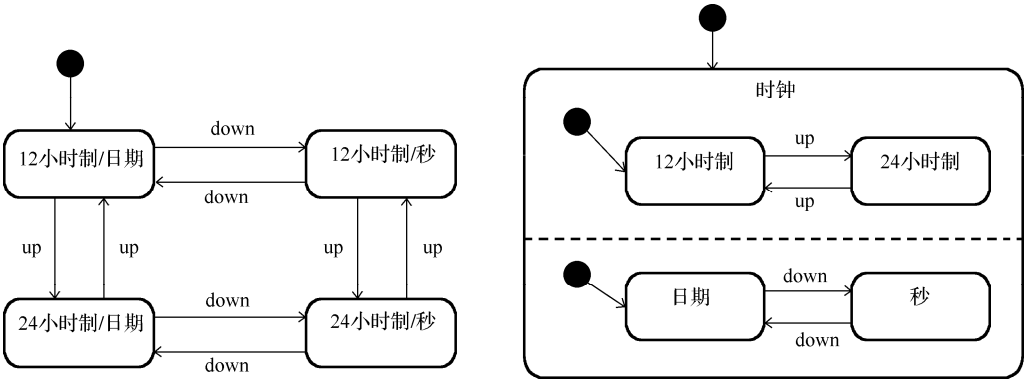


图 7.9　状态图中的并行结构

尽管复合状态和并行子图的引入减轻了状态图的描述负担，但在对业务行为进行设计时，还要针对具体需求考虑，以下几条规则可以作为参考：

(1) 如果一个事件在两个子状态图中都需要进行处理，为两个子状态分别建立一个到各自结束状态的转换。

(2) 从含有并行子状态图的复合状态出发的转换表示，只有当其含有的所有子状态图都位于结束状态时，该转换才会被触发。

(3) 若某个复合状态中的一个子状态图的分支导致其离开了此复合状态，则所有的子状态图都将结束。

7.3.3　状态图的应用

在软件开发的过程中，根据不同的开发系统类型，状态图会有不同的作用。对于经济领域中的软件系统，如企业资源计划(ERP)系统和作为演示例子的“软件项目管理系统”，状态图并不是很重要，因为相对来说，这些系统中具有较少的复杂对象，而状态图比较适用于对复杂对象的行为描述。

在前面的示例系统中，一个对象包含的方法几乎可以在任何时刻进行调用。例如，对于实体类 Employee，其修改员工名字或地址的方法一般可以不依赖参数值进行独立的调用。但是，若系统中存在单独的业务中心类，则对这些类使用状态图进行规格说明将会十分有用，如图 7.8 中的状态变化就是这样的情况。状态图可以较清晰地描述出它们的行为，如什么时间、哪些方法可以进行调用的情况一目了然，其他类似的还有订单、商品购物清单或者图书的借阅

状态等。若在一个状态中某个不允许的方法被调用，则要有一个错误处理的过程与其对应，它们可以是简单的忽略或者详细的异常处理，例如，项目若没有结束其项目计划状态，则只能进行项目取消的操作。

基于事件驱动的状态图非常适合硬件或嵌入式系统的行为建模，因为它们通常是通过信号(Signal)进行通信的，信号可以理解为输入的事件，然后在动作部分或者状态描述中确定需要产生哪些新的事件。这样的系统是典型的被动式(Reactive)系统，图 7.10 给出了一个电机控制系统的构成图，主要包括启动器、离合器、自动控制器、启停控制器和电机控制器。

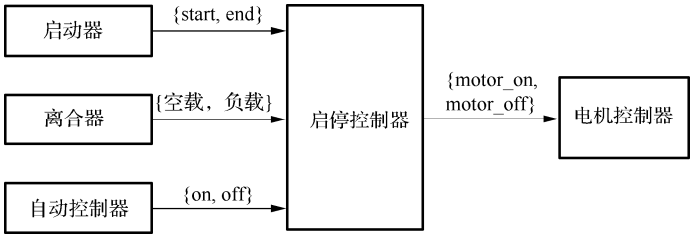


图 7.10　电机自动启停控制系统的构成

各装置之间的箭头标明了相关事件的名字，箭头方向表示信号传递的方向。具体地，启动器能够发出 start 和 end 信号，用于对系统加电或断电；自动控制器能够发出 on 和 off 信号，用于设置当前电机工作模式；离合器能够发出空载或者负载的信号。电机外部的温度通过传感器连续地进行测量，测量值保存在本地寄存器中。启停控制器会产生信号 motor_on 和 motor_off，驱动电机控制器运转。

图 7.11 描述了这个电机自动启停控制系统的状态图。首先，由启动器发出 start 信号进行加电，此时自动控制器位于关闭状态。在没有自动控制的情况下，离合器产生负载，系统发出 motor_on 动作，使电机处于行进状态。如果自动控制器从 off 置为 on，会使得行进中的电机继续运转，但处于自动启停模式。这里，电机有可能连续多次收到 motor_on 信号，而中间没有 motor_off 信号，这在设计和开发控制器时需要在行为上相符合。当电机在自动启停模式下时，若空载并且电机外部温度在 3～30℃，则会进入待机节能状态。在工作的任何状态，如果启动器发出 end 信号断电，那么会使得电机停止运转。

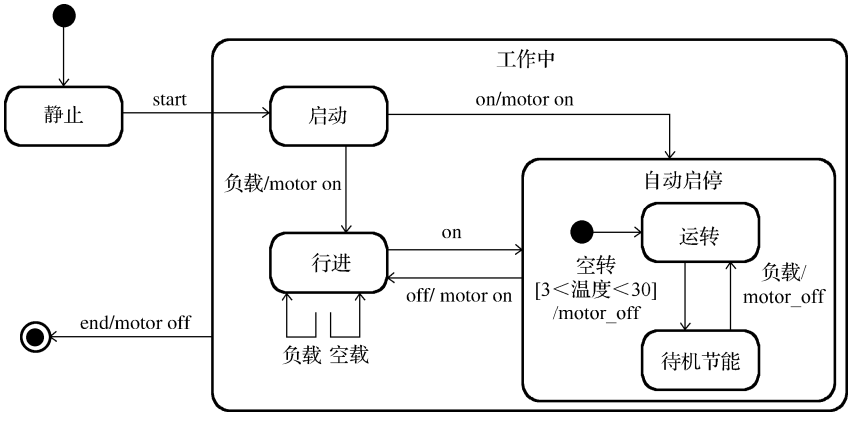


图 7.11　电机自动启停控制系统状态图

除了类似的硬件系统，应用状态图来描述用户界面的控制也较为常见。这里的状态用来

表示每个独立的图形界面，事件是用户的输入，如某个按键被按下，进而迁移到另外一个界面，即到达了一个新的状态。图 7.12 给出了一个这样的示例，应用一开始从 start 状态开始，对应的是一个初始界面，当用户输入了用户名和密码并单击“登录”按钮时，导致另外一个事件发生，但具体发生的事件依赖程序内部的验证结果：若登录不成功，则显示登录失败界面，通过单击“回到开始”按钮，返回到初始界面；若登录成功，用户将会进入一个复合状态“应用运行”中，这里含有若干与业务相关的其他状态，图中只是给出了简单的示意性描述，层次结构也是用来帮助理解的。该状态还提供了一个带有向外的“退出”事件的转换，表示任何在“应用运行”状态中的子状态都能够响应该事件并退出系统，回到登录的初始界面，这种描述方式使得这些子状态能够保持统一的行为。

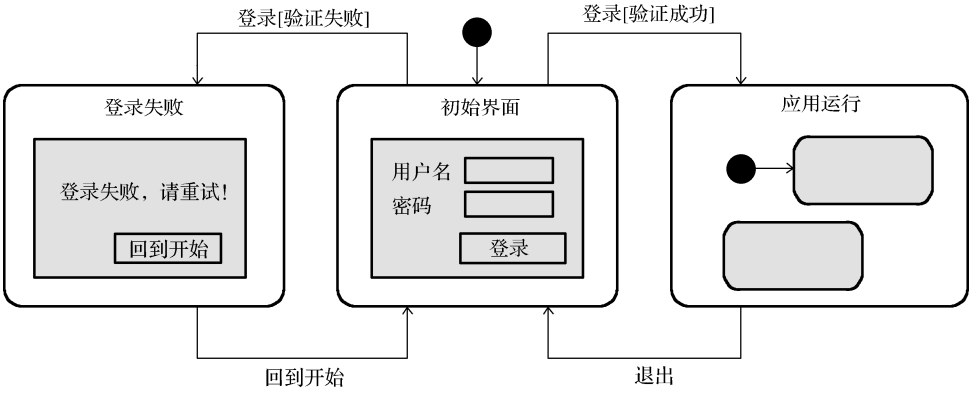


图 7.12　描述 GUI 界面的状态图

7.4　对象约束语言

以类图为主的建模过程经常会遇到很多需求和规则无法利用现有的模型进行描述的情况，一个典型的例子是项目类不能以自身作为其子项目的要求，这是由于图形本身的局限性造成的。UML 提供了一种对每个 UML 对象进行条件约束的机制，并通过将约束条件放置于花括号中进行表达。例如，不允许项目编号这个属性的取值为负值，可以在类图中对该实例变量进行约束：

```
- projectnumber: int {projectnumber >= 0}
```

但是，这些补充的内容会使得类图的可阅读性变差，并且使得其与其他对象间的复杂关系不能通过一种尽量简洁的形式表述出来。基于这个原因，UML 使用一种标准的对象约束语言(Object Constraint Language, OCL)来对类和对象所依附的条件进行正式定义。

本节借助一个新的类图进行说明，如图 7.13 所示。这里有一个学生类 Student，其对象含有一些简单的属性，如姓名、学号、专业及是否休学期等。通过属性“所选课程”，能够确定出该学生当前学习的课程或者已经学过的课程。关联关系“选课”是一个双向导航的关系，所以在课程类中也具有一个实例变量，记录着每门课程至少有三名选课学生。每门课程除了课程名外，还有一个课程状态属性，通过一个枚举类型进行定义。

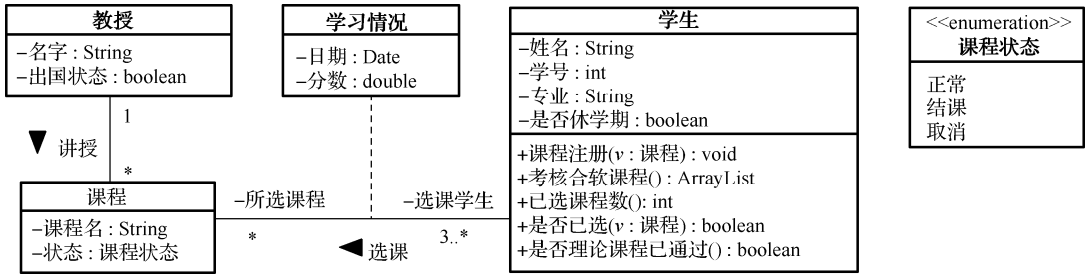


图 7.13　OCL 示例类图

在这个类图中还可以看到一个新加入的表示方法，即对关联关系的属性进行更详细的描述：这里有一个单独的关联类“学习情况”，与所属的关联关系通过一条虚线相接。这些关联类在形式上是普通的类，可以像使用普通类那样使用。但是，需要注意的是，每个关联对象只与其对应的一个学生对象和一个课程对象具有联系。对于一个关联关系来说，是否具有关联类是可选的内容。当然在此例中，这个关联类的存在还是有必要的，因为它对所属关联关系的特征进行了较好的描述。

另外，图 7.13 中还存在一个教授类 Prof，其与课程之间存在一对多的讲授联系。也就是说，一名教授可以讲授多门课程。该类中还有一个属性，用来记录本学期该教授的出国状态。

如果要求学号长度至少为五位数字，并且构成的学号要大于或等于 10000，若使用 OCL 进行约束，则可以写成如下形式：

```
context Student inv regStudentId:
   self.stuId >= 10000
```

通过关键字 context 表明这段 OCL 的描述与哪些 UML 对象或类相关。关键字 inv 表示这是不变的(invariant)，也就是说，所有的 Student 对象都要遵守这个约束。关键字 inv 后面的内容是可选的，表示 OCL 约束的名字，大多数情况下可以省略，但最后的冒号必须存在。

context 描述的内容确定了约束对象的范围，这里的 self 关键字指的就是对这些对象的引用，正如 Java 中的 this 或 C++中的*this 一样。在规则的描述中，可以直接对所有的实例变量和方法进行访问。如果对 self 的使用存在某些理解上的障碍，可以为目标对象赋予一个名字。下面的示例给出了类似的实现，其语义与前面的示例等价：

```
context s:Student inv:
   s.stuId >= 10000
```

由于精确的规格描述经常会需要很多 OCL 共同进行约束，这些约束往往以包中的类为单位进行分配，因此可以将 OCL 以包的形式进行组织，它们的结构可以按照对应包的包结构那样有层次地组织起来，OCL 的语法如下：

```
package com::myCompany::mySW
   context Student inv:
   context Student inv:
endpackage
```

除了对实例变量的约束，OCL 同样可以对实例的方法进行约束。这里并不是要去描述其

具体的实现内容，而是针对这些方法的前置条件或者后置条件进行约束。例如，对于约束“未在休学期的学生必须注册一门课程”。OCL 可描述为如下形式：

```
context Student::selectedLectures(): Integer
   pre stustatus: self.freesemester = false
   post selectedLectures: result > 0
```

从 context 中可以看出，此约束是针对 Student 类的 selectedLecutres()方法的，这种标记方式类似 C++中的方法实现，还给出了此方法的返回值类型。关键字 pre 后的内容表示前置条件，其名字为 stustatus，名字为可选内容。前置条件明确了只有当该条件为真时，此约束才会产生作用。关键字 post 后面紧跟的是后置条件的说明，其名字 selectedLectures 也是可以省略的。OCL 关键字 result 是指此方法的结果，即要求返回值大于零。

只能对实例变量或返回值在约束中进行访问，有时是满足不了要求的。对于约束，OCL 描述如下：“当该学生选择了一门之前没有学过的课程时，该学生选课总数就加一。”

```
context Student::registerLecture(v: Lecture)
   pre: not isSelected(v)
   post: self.selectedLectures()@pre = self.selectedLectures() - 1
```


registerLecture 方法中含有的参数也一并在 context 中接收过来，而且在前置和后置条件中对目标对象中的其他方法都可以进行引用。需要注意的是，这些方法使用时是在 context 中给定的方法 registerLecture 执行之前还是之后。如果没有特殊说明，默认指与在该方法执行之后的状况相关联。若需要执行之前的状态值，则需要如上述例子中那样，在方法的后面加上@pre 加以说明。

OCL 首先支持表 7.3 所示的简单类型，并且同时给出了几个对应的方法。OCL 能够支持的其他类型包括一些扩展类型，如集合类型 Collection 等。

表 7.3　OCL 中的常用基本类型及相关方法

类 型 名	取 值 示 例	相 关 方 法
Boolean	true, false	and, or, xor, not implies, if then else endif
Integer	1, -2, 50, 464646	*, +, –, /, abs()
Real	3.14, 42.42, -99.99	*, +, –, /, floor()
String	‘Hello’, ‘Dalian’, ‘’	concat(), size(), substring()

对于实例变量的访问，一般通过引用变量名字来实现。当类图中不存在某些角色名称时，对这些变量的引用可以通过其关联类进行，类名在这里要写成小写形式。这个约定在单向导航的关联关系中非常有用，例如，用在关联关系的相反方向需要进行约束的要求，实际上是很常见的情况。举例来说，一个项目任务一定不需要知道其所属的项目，但对于项目任务来说，其计算出的工作量一定要小于等于所属项目总的工作量，因此这样的约束描述是很有现实意义的。又如，某门课的授课教授在上课时其出国状态不能为真：

```
context Lecture inv:
   self.status = Lecturestatus::running
   implies not self.prof.inForeign
```

关键字 implies 表示“if-then”的条件说明。另外，对教授对象的访问通过 self.prof 的形式，而且在 OCL 中通过句点的形式对实例变量的直接访问是永远有效的，如同对某个枚举类型值的引用。

通过类名的访问方式对于关联类也适用，而且它们可以通过连接对象的角色名字进行引用，如对于一门结课的课程成绩要求在 1～5 之间，可以进行如下描述：

```
context Examination inv:
    self.selectedLecture.status = Lecturestatus::closed
    implies (self.note >= 1.0 and self.note <= 5.0)
```

到目前为止，我们只关注了实例变量的各种约束方法，它们通常只对一个简单值进行存储。通过使用 OCL，还可以对集合类型进行约束。OCL 中目前只支持一些标准的集合类型，如 Collection、Set、OrderedSet、Sequence 及 Bag。集合类型的实例变量可以使用 OCL 中预定义的方法，如查找单一元素、查找多个元素、是否存在元素、是否具有某个特定的属性或者对集合类型的遍历等，它们与编程语言的类库中针对集合类型提供的方法类似。如果要使用某个集合 collection 的方法，可以通过如下的 OCL 表达：

```
collection -> method(parameter)
```


例如，一名学生在一个学期内最多可以选择 12 门课程的约束，可以通过如下方式进行描述：

```
context Student inv:
    Student.selectedLecture
        -> select (s | s.status = Lecturestatus::running)
        -> size() <= 12
```

通过 Student.selectedLecture 返回该学生所选所有课程的集合，然后该集合的 select 函数被执行，返回的也是集合类型。具体地，select 函数会根据输入的过滤条件对该集合中的数据进行筛选，那些正常开课状态的课程将会作为结果返回。该方法参数中竖线前的变量 s 可以理解为一个循环变量，用于循环遍历集合 student.selectedLecture 中的每个元素，也可以将 s 理解为 Java 或 C++标准类库中的迭代器。返回的集合又进一步使用了 size()方法，其提供集合所含元素数量的整型值，这里的约束需求为小于等于 12。

若在已考试的课程中存在“理论”课程，则方法 hasTheoryLect()返回真值。此约束可按照如下的形式进行描述：

```
context Student::hasTheoryLect():Boolean
    post: result = self.examination
        -> exists( p | p.note >= 60 and p.selectedLecture.title='Theory')
```

通过 self.examination，可以得到所有与该学生相关的考试对象。通过学生对象，可以访问其所属的所有关联对象。通过 exists 遍历集合中的每个元素，并当其中至少一个元素满足 exists 中的指定条件时，返回真值。

还可以使用 OCL 对方法的返回集合类型进行约束。如要求方法 passedLectures()返回的所有课程必须是已经学过的课程，此约束对应的 OCL 可以按照如下方式进行描述：

```
context Student::passedLectures():Collection
    post: result = self.examination
        -> select( p | p.note>=60) -> iterate(p:Examination; res: Collection
= Collection{} | res->including(p.selectedLecture))
```

首先，通过 self.examination 返回 Student 所属的所有考试对象，然后留下那些考试及格的对象。接下来，必须从这些对象出发与该学生所修课程进行对比。这里使用了集合对象具有的 iterate 方法，其具有两个必要的参数：一是一个枚举变量，用来遍历集合当中的每个元素，其后面可以是该变量的具体类型；二是对集合的遍历结果，在指定其类型的同时必须还要给定一个初始值，因为不同的集合类型在迭代中可能会存在多种不同的计算方式。这个例子中的结果变量名为 res，类型为 Collection 且初始值为空。该条件指定了 res 在迭代后会含有与每个通过的考试对象对应的所有课程。方法 including 仅当给定的参数对象在该集合中出现时才会返回真值。

OCL 是一种条件约束语言，因此并不能直接从 OCL 翻译到具体的代码实现；OCL 是一种描述性语言，因此在建模过程中可根据情况进行指定，并不需要强制给出。

7.5 习题

1．给出下面代码段的程序盒图，并使用状态转换图对该代码段中变量 n 的状态变化进行描述。

```
n=0;
for (i=0; 0<10; i++)
    for (j=0; j<i; j++){
        n=n*j;
        switch(n){
            case 0: n=2; break;
            case 1:
            case 2: n=n%8; break;
            case 3:
            case 5: n=n+3; break;
            default: break;
        }
    }
```

2．IP 长途卡的使用过程如下：未使用前，卡密封在塑料卡袋中，密码被遮挡住；一旦开始使用，使用者必须打开卡袋、刮开密码，通过座机绑定使用长途卡；一旦卡上余额不足或者超出长途卡的使用日期范围，长途卡将无法使用。根据以上描述，分析 IP 长途卡的状态，绘制其状态图。

3．绘制该伪码对应的程序流程图。

```
Input n; //输入数组大小
```

```
Input List; //从小到大输入 n 元有序数组
Input Item; //输入待查找项
Start = 0;
Finish = n-1;
Flag = -1;
while (Finish - Start > 1 && Flag == -1){
    i = (Start + Finish)/2;
    if(List(i) == Item) Flag = 1;
    else if(List(i) < Item) Start = i + 1;
    else Finish = i - 1;
}
if (Flag == -1){
    if(List(Start) == Item) Flag = 1;
    else if(List(Finish) == Item) Flag = 1;
    else Flag = 0;
}
Output(Flag);
```


4．某公司承担空中和地面运输业务。计算货物托运费的规定如下。

[空运]若货物重量小于等于 2kg，则一律收费 6 元；若货物重量大于 2kg 而又小于等于 20kg，则收费 3 元/kg；若货物重量大于 20kg，则收费 4 元/kg。

[地运]若为慢件，则收费为 1 元/kg。若为快件，当重量小于等于 20kg 时，收费为 2 元/kg；当货物重量大于 20kg 时，收费为 3 元/kg。

请画出以上规则的判定树。

5．绘制一个描述鼠标单击事件的状态图，将间隔时间或发生的次数作为条件和动作执行的依据。当一个对象被单击选中并且存储在一个变量中时，若在 0.5s 内此对象又被单击，则被认为是一次双击操作；若单击的是另外的对象或者该对象在至少 0.5s 后才被再次单击，则被认为是一次对该对象的单击操作。

6．画出下面用 PDL 写出的程序的 PAD 图。

```
while P do
    if A>0
        then A1
        else A2
    end if;
    S1;
    if B>0
        then B1;
            while C do
                S2; S3;
            end while;
        else B2;
```

```
        end if;
        B3;
    end while;
```

7．基于类图 5.9，使用 OCL 完善以下约束：

(1)每个项目的计划开始时间不得晚于 2018 年 9 月 5 日。

(2)每个项目不能作为其本身的前驱项目添加到前驱项目列表中。

(3)每个项目的任务集合中不能同时有超过 3 个完成进度不足 50%的任务。

第8章 设 计 优 化

前面的内容主要围绕类图模型的构建展开，并通过状态图对类的具体行为进行设计，使用 OCL 约束机制在类图中补充较复杂的业务规则，这些内容都是代码实现的重要依据，并保证物理层面能够高质量地体现和忠于设计思想。

这里说的高质量，一方面要求忠于设计，另一方面要能够尽可能地适应未来的变化，并且在设计中留有余地，做到以不变应万变。当然，绝对应万变的设计是不存在的，我们可以做的是尽可能提升设计方案对变化的适应能力，这也是本章主要讨论的内容。

本章从设计原则的角度出发，它是面向对象设计优化的基础。细节决定成败，但注重细节不等于只见树木不见森林，要在细节中体现出设计的质感和层次感。高质量的建模要以简单和小为基本观点，从方法级开始，逐渐过渡到组件、架构及更高级别的组织策略，对设计原则进行体现。

本章还讨论设计模式，设计模式是随着面向对象技术的经验积累与逐渐成熟，人们在开发实践中认识到一些经常发生的问题具有相同或相似的解决方案，并在此基础上达成的共识。本章将结合具体的例子对一些常用的设计模式进行介绍，如抽象工厂等设计模式。

8.1 小即是美

在软件设计领域里，小的设计强调的是一种恰如其分的设计哲学。这里的“小”是指在开发过程中，每一次迭代的目标不宜设立过大，需小步前行，避免过度设计。在设计时，整个系统最好由松散耦合的细小模块组成。这些细小模块由于功能相对独立、单一，因此更易于理解。

“小即是美”体现出设计思想的灵活性、完整性与轻量性。灵活性在于它能快速地响应变化，这种变化可能是局部的，也可能是整个设计方向的改变。一旦需求要求我们对设计做出改变，我们也能将修改控制在足够小的范围，从而保证对整个系统不会带来巨大的影响。完整性在于它不是残缺的，但也并不意味着大而全，而在于它足够精简，没有冗余。轻量性体现在它的功能并不臃肿，对外部的依赖较少，既容易在系统中快速引入，又不会使原有系统变得笨重，还能很方便地部署或者启动。

小规模设计是设计优化的基础。这里存在两个一般性的原则可以作为参考。

KISS（Keep It Simple Stupid）：应该选择那些尽可能简单的实现方案，因为它们既能够全面解决问题，且具有较好的可理解性。这并不意味着“Quick and Dirty”方式，而是对简单的开发方式的一种认可。

YAGNI（You Ain't Gonna Need It）：这条原则的含义是“你不会需要它”，是指开发者自以为有用的功能，实际上都不会需要。也就是说，我们不应该过多地从理论上考虑未来可能的扩展而对设计进行一厢情愿的泛化改进，例如，并不是每个集合类型的实例变量都需要 add 和 remove 方法。

追求简单和小的设计使得目标系统在开发过程中逐渐趋于清晰和简洁，好的设计规则的应用能够带来简化的设计和可维护性高的系统，反之会带来冗杂的系统设计和可维护性差的系统。对业务缺乏洞察、设计的随意性和不好的习惯会使设计和实现变得很差，这些缺陷在软件工程领域又称为设计的“味道(smell)”，它们通常具有如下特点。

(1)僵化性：隐藏的设计关联性导致对文档或代码进行小的修正时会埋下不可预期的隐患，导致系统的修改困难重重。

(2)脆弱性：在进行一个修改时，导致程序许多没有概念关联的地方出现问题。

(3)顽固性：分离设计中包含的有价值的部分并进行重用的付出和风险是巨大的。

(4)粘滞性：开发者在添加新功能时只是在现有代码的基础上拼凑，不愿意触碰现有代码，不对代码重构，导致原有设计的毁坏和退化。

(5)不必要的复杂性：设计人员预测需求的变化，为过多的可能性做准备，致使设计中含有绝不会用到的内容，无法带来回报。

(6)不必要的重复性：设计中含有重复的内容，而这些重复的内容本可以使用单一的抽象进行统一，导致修改时无法保持一致。复制和粘贴的编码习惯是导致大量重复代码的主要根源。

(7)晦涩性：模块难以阅读和理解，代码随着时间而演化，变得越来越晦涩，逐渐丧失清晰性和表达力。

下面将从实际例子出发，再次说明设计优化的必要性及其作用，进而为一些重要设计和优化原则做出铺垫。

8.2 设计优化思想

在需求分析和类的概要设计中，主要以业务逻辑为主线，给出分析和设计类图。类图描述了将来要实现的业务目标，而面向对象设计阶段的主要工作是从分析类图向程序实现过渡，这个阶段主要考虑的是如何丰富类图的内容，以提供更贴近实现的策略。在这个过程中要注重类图结构的稳定性，以灵活应对未来需求的变更。

8.2.1 运行时的多态

多态的基本思想是：对类中每个变量，在保证业务成功实现的前提下，尽量保持类型泛化的定义。多态在结构上其实就是形成类的继承层次，如 B 类从 A 类继承，C 类从 B 类继承，在对具体变量进行使用时，根据需要将某个变量定义为根类型 A 类就足够了。这时，此变量对应的实际类型可能为 C 类，但按照 A 类使用，则只有 A 类中的方法是可用的。当然，A 类中方法的具体行为是可以在 C 类中重写的，重写的具体要求如下。

(1)重写的方法本质上与父类方法具有相似的行为，但在细节上进行了有针对性的调整。

(2)重写的方法与原方法在相同的条件作用下工作，子类方法不应具有比其父类更严格的条件限制。

(3)重写的方法不能超出父类方法的状态。

根据以上要求，若一个方法的输入范围是 1～10，返回结果的范围是 40～60，则对其进

行重写的方法的输入范围需要包含原输入区间，如–10～10；结果输出区间应是原输出区间的子集，如 43～55。

接口的使用能够更容易地说明多态，如图 8.1 中的接口 GeoObject，此接口衍生出三个具体类的实现，每个类对于接口中的方法具有不同实现方式。Point 类和 Rectangle 类中的构造方法的参数使用了默认值的描述。在该类的构造过程中，可以不具体指定相关参数的内容，该类会自动按照其默认值进行构造。在 C++中，方法的声明部分可以直接指定构造函数的默认值，但在 Java 中不允许使用这种方式。因此，在 Java 中，要么把三种构造情况都写出来(对应三个构造函数)，要么使用 Java 5 版本以后提供的可变长列表的类型。代码 8.1 为 Point 类对应的 Java 实现。

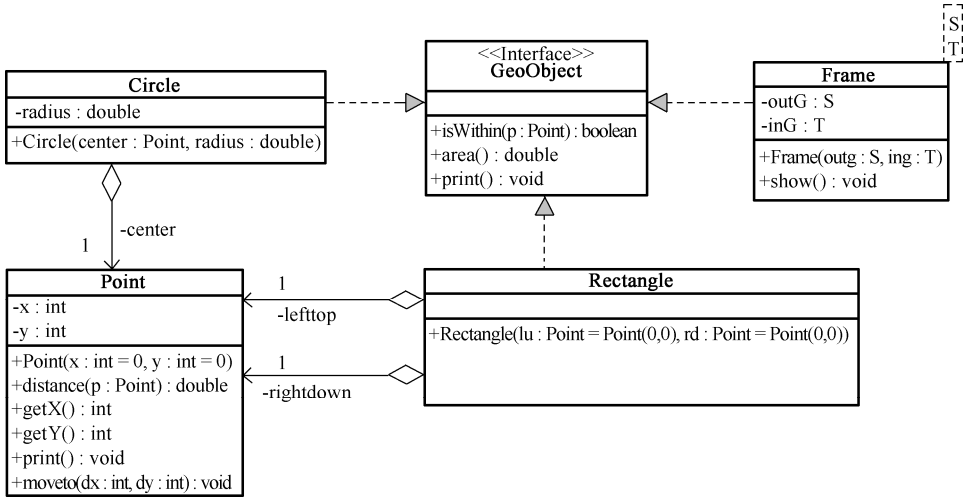


图 8.1　通过接口实现的多态

代码 8.1　Point 类的 Java 实现

```
public class Point {
    private int x = 0; //default value
    private int y = 0; //default value
    public Point(int... x){
        if(x.length > 0)
            this.x = x[0];
        if(x.length > 1)
            this.y = x[1];
    }
    public static void main(String[] s){ //for testing
        Point p1=new Point();
        Point p2=new Point(4);
        Point p3=new Point(5,42);
    }
}
```

代码 8.2 对图 8.1 中与接口 GeoObject 相关的类创建了对应的对象，主要关注它们的初始化及访问方式。

代码 8.2　GeoObject 接口多态的实现

```
public class Main {
    public static void main(String[] args) {
        Rectangle r1=new Rectangle(new Point(6,16),new Point(27,3));
        Rectangle r2=new Rectangle(new Point(12,13),new Point(19.5));
        Circle k1=new Circle(new Point(15,9),8);
        Circle k2=new Circle(new Point(15,9),4);
        GeoObject[] g={r1,k1,r2,k2,
               new Frame<Rectangle,Rectangle>(r1,r2),
               new Frame<Rectangle,Circle>(r1,k2),
               new Frame<Circle,Rectangle>(k1,r2),
               new Frame<Circle,Circle>(k1,k2)};
        for(GeoObject obj:g)
           System.out.println(obj.getClass() + "\t: " + obj.area());
    }
}
```


运行时的多态机制，使得对对象变量的某种方法的调用是动态的，并与该对象实际类型中对应的方法进行绑定，也就是说，运行时实际执行的方法是该对象实际类型中的方法。利用好运行时的多态性，可减少 switch(或 case)语句的使用，switch 语句的问题在于它很容易带来分散、重复，但又相同的计算逻辑，多态提供了一种优雅的解决办法。

8.2.2　耦合的消息链

在分析和设计阶段，一个首要的问题就是确定功能方法的实现位置，例如，在概要设计类图中，可以通过管理类实现对实体类的管理和访问。因此，如果要对 ProjectTask 类的完成进度进行修改，那么需要管理类 ProjectMan 首先委托相关的项目对象查询目标任务，然后再进一步修改该任务的完成进度，这个过程可由图 8.2 所示的顺序图进行表示，图中假设第 i 个项目任务为 pa，对应的代码是一个耦合消息链(Message Chain)：

```
this.getSelectedProject().getTask(i).setCompeletePct(50)
```

这种设计方式的缺点是 ProjectMan 类不仅要了解其自身的实例变量属性，同时也要熟悉其他已知类的实例变量，即需要了解所有以各种形式参与到场景中的对象，这增加了对象间交互的复杂程度，不利于设计的模块化组织。

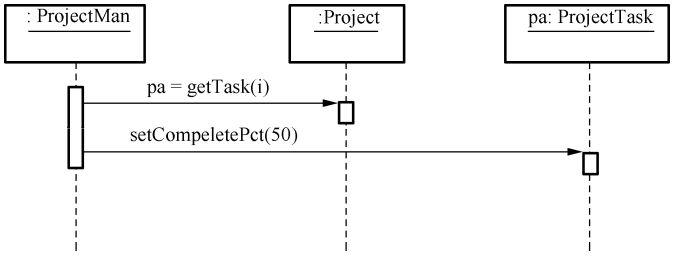


图 8.2　交互集中的设计

改进方法的思想就是减轻管理类的负担，将不适合 ProjectMan 对象本身直接处理的信息通过一个实例变量委托至另外一个更合适的 Project 对象进行处理，这时几个对象的交互过程如图 8.3 所示。这种方法的好处是每个类只需了解与其本身相关的子任务，避免了不得不去了解其他所有类的情况，分散的交互方式使模块的组织简单化。

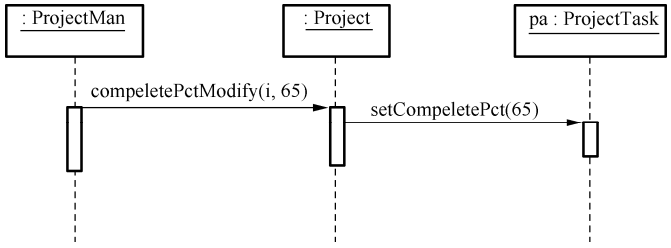


图 8.3　交互分散的设计

8.2.3　狎昵关系

上述例子中的主要要求就是类的功能应尽量在一个类中进行实现，使得类不需要了解更多的信息，就能够完成该项工作，降低了类与类之间的耦合程度。接下来，再讨论一种狎昵关系(Inappropriate Intimacy)的设计味道。这是一个多米诺骨牌的游戏：每枚多米诺骨牌有左右两个数字，出牌时要求前一张牌与后一张牌首尾数字相同。设计上可设置一个 Domino 类，并通过两个实例变量(左值和右值)分别表示两个牌面。另外，游戏控制类 Game 负责控制游戏的进行，其中含有一个 Domino 类的实例变量 middle 用来记录当前可以连接的左、右牌值，简单的类图如图 8.4 所示。

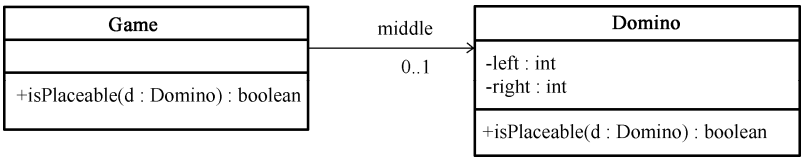


图 8.4　多米诺骨牌游戏类图

对于游戏规则的实现，一种方式是出牌时向 Game 对象传递出牌对象，并判断此牌是否可以在首或尾衔接。对应的 Java 实现如代码 8.3。

代码 8.3　Java 实现 1

```
//in Game
public boolean isPlaceable(Domino d){
    return d.getLeft() == middle.getRight()
                                || d.getRight() == middle.getLeft();
}
```

这种设计的主要问题是：Game 对象需要深入 Domino 对象的内部以获得需要的牌面信息，无论是传递进来的参数 d 还是 middle 对象，这个方法高度关注 Domino 类的私有成分，导致了两个类之间过分亲密的关系(狎昵关系)，即高耦合。由此可见，面向对象提供的封装机制并不能完全保证模块的独立性，必须要在设计中充分考虑业务结构的特点，并做出利于稳定的设

计决策。因此，这里优化的目标就是阻止两个类间的狎昵关系，采用代码 8.4 的实现方式，由 Domino 对象本身来判断衔接位置的合规性，同时避免了在 Game 类或其他类中的对出牌规则的判断多次重复实现。

代码 8.4　Java 实现 2

```
//in Game Class
public boolean isPlaceable(Domino d){
    return middle.isPlaceable(d);
}
//in Domino Class
public boolean isPlaceable(Domino d){
    return left == d.getRight() || right == d.getLeft();
}
```

8.2.4　被拒绝的遗赠

如果已经存在某个可用的类，但是该类不具有所有需要的功能，为了满足需求，需要在该类的基础上进行功能和结构的扩展，这也涉及对现有类的重构。有两种方法可以作为调整类设计结构的参考。

第一种方法是借助继承的方式。从已有的类继承新的类，在新类中扩展原有的功能，并与父类提供的方法一起完成所有需要的功能。这种方法适合已有的类是自己开发完成的，或者是基于一个框架系统(见章节 9.4.2)开发的。但这种方式需要注意一些特殊情况，例如，子类只是使用了父类的一部分属性和方法，父类的其余部分子类是不需要的。尤其是当子类只是需要重用父类的某些功能，但对于父类的行为并不想保持一致时，就产生了被拒绝的遗赠现象(Refused Bequest)，它往往预示着继承层次的误用，这时应考虑使用第二种方法进行扩展。

第二种方法通过委托(Delegation)进行，其基本形式是新类与原有类构成关联关系，即具有一个它的实例变量。新类可以对所有已有类中的公开方法进行调用(消息发送)，由于是一种转发方式，所以在性能上可能会存在影响，但是新的功能会非常容易地在新类中进行重用，而且与上层类之间的依赖也不再强烈。

8.2.5　循环依赖

两个类分别处于不同的包中，并且具有双向导航的关联关系，这样的循环依赖是不可避免的。如果找不出更简单的设计，为了消除类似的循环依赖，建议按照图 8.5 所示的方式进行设计结构的优化，从而消除循环依赖。图 8.5 的左侧是两个类 A1 和 B1 具有的双向关联的设计图，A1 需要 B1，同时 B1 也需要 A1，从而导致了两个包之间的相互依赖。图 8.5 的右侧部分是在左侧的基础上，通过一个接口的引入，巧妙地消除了循环依赖的存在。

具体方法是：在包 A 中添加了对 B1 的接口 IB1，该接口中含有所有 A1 需要的 B1 方法的定义。这样设置后，A1 不会直接依赖 B1，而是使用了其接口 IB1，而 IB1 位于包 A 中，这样就打破了两个包之间的循环依赖状态，整体上只保留了包 B 对包 A 的依赖，这是一种设计原则的体现。

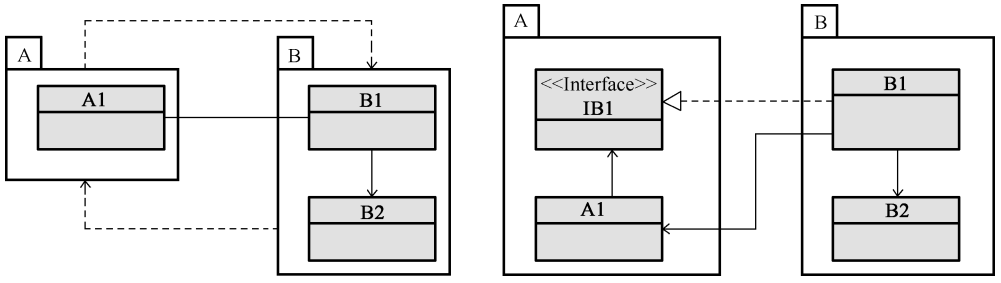


图 8.5　循环依赖的消除

8.3　设计原则

类图是实现的参考，而软件设计又是一门艺术，是对“变化”的辩证处理，即发现变化，隔离变化，以不变应万变。本节在上节的基础上，将常用的设计思想进行提炼和归纳，产生最基本的几个设计原则，它们是对面向对象思想的深化和具体体现，它们的综合应用是产生高质量软件设计的基石。

8.3.1　接口隔离原则

接口隔离原则(The Interface Segregation Principle，ISP)有两层含义：

(1)应尽量使用“接口继承”[①]，而非“实现继承”。接口关注对象的概貌，将对象中“不变”的信息抽象出来，不涉及细节，因此是“稳定”的。

(2)通过接口，只将需要的操作“暴露”给需要的类，而将不需要的操作隐藏起来。接口在这里充当类的视图。

图 8.6 描述了一个遵照 ISP 的设计，IManeuverable 是具体交通工具(如 Car、Boat、Submarine 等)的抽象，它提供了驾驶的接口，如加速、减速、转弯等。Client 类在驾驶(drive)交通工具时，使用的不是具体类，而是其接口类。这样做的好处是：当业务需求变化时，更容易发生改变的是具体类，而这些变更可以通过稳定的抽象类进行隔离，使得 Client 类不受变化的影响，从而提高了系统的可维护性。

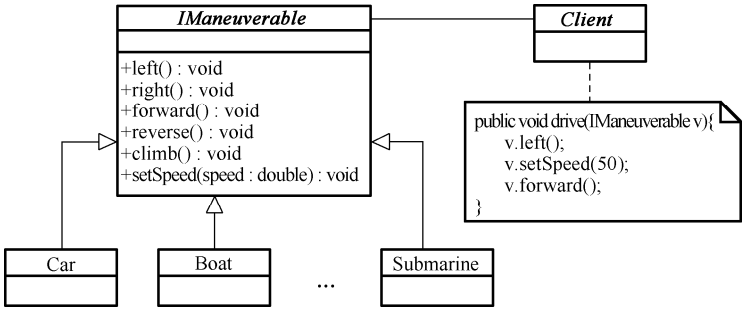


图 8.6　ISP 实例 1

① 这里的接口指的是广义上的抽象类，可以是面向对象编程语言中的 interface 或 abstract class。C++通过继承纯虚类来实现接口继承；Java 对接口继承具有单独的语言构造方式。

上例中的 Client 类专注于驾驶，它对交通工具运转的机理是不关心的，因此通过接口 IManeuverable 暴露给它的只是与驾驶相关的操作。如果系统中还存在另一类使用者，如维修者(Repairer)，它们专注的应该是这些交通工具的运转工况，而与驾驶相关的操作不应该暴露给这些维修者，如图 8.7 所示。

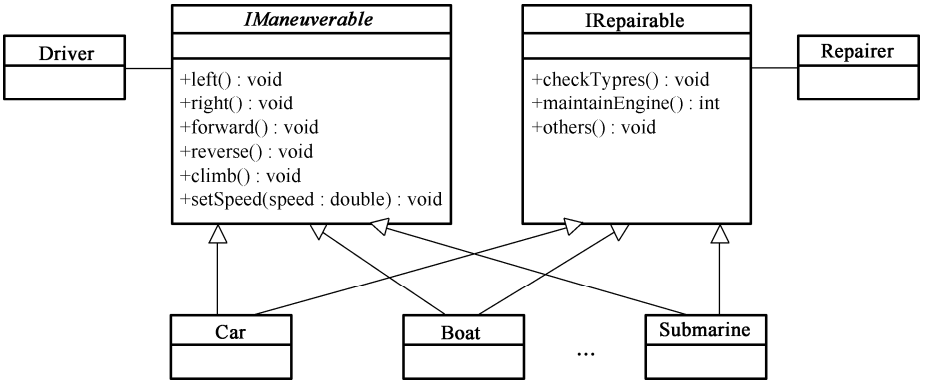


图 8.7　ISP 实例 2

面向接口的设计能够使 Client 只需关注如何进行业务活动(如驾驶)，而不必关心其使用对象的具体实现。一个对象可以很容易地被(实现了相同接口的)另一个对象所替换，这样对象间的连接不必硬绑定(Hard Wire)到一个具体类的对象上，因此增加了灵活性。这是一种松散的耦合，同时增加了重用的可能性。

8.3.2　依赖倒置原则

依赖倒置原则(Dependency Inversion principle，DIP)的宗旨是应依赖于抽象，而不要依赖于具体。我们在 ISP 中已经提到，抽象(接口)描述的是对象的概貌，而这种概貌是我们从现实世界普遍规律中提炼出来的，因此能够做到最大限度的稳定。所以很容易从抽象对对象进行扩展，可以这样认为：扩展的基础越具体，扩展的难度就越大，具体类的变化无常势必造成扩展类的不稳定。

依赖倒置原则使细节和具体实现都依赖于抽象，抽象的稳定性决定了系统的稳定性。从物理上也可以这样解释，如图 8.8 所示，一个基础稳定的系统要比一个基础不稳定的系统在整体上更“稳定”一些。

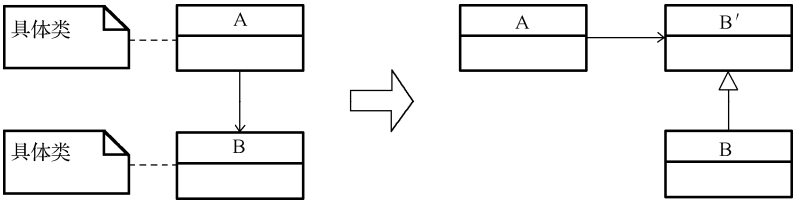


图 8.8　依赖倒置原则

8.3.3　开放封闭原则

开放封闭原则(The Open Closed Principle，OCP)是指一个模块对于扩展应该是开放的，而

对于修改则应是封闭的。这条原则是面向对象思想的最高境界。简单来说，就是开发者应给出对于需求变化进行扩展的模块，而永远不需要改写已经实现的内部代码或逻辑。具体地，它有两个基本的特点：

(1) 模块的行为可以被扩展，以满足新的需求；

(2) 模块的源代码不需要进行修改。

从上面的分析中可知，OCP 是相对的，没有绝对符合 OCP 的设计，而且一个软件系统的所有模块不可能都满足 OCP，我们需要做的是要尽量最小化这些不满足 OCP 的模块数量。例如，图 8.9 给出了一个满足 OCP 的设计，Client 类很容易地计算所有 Part(零件)的价格之和，并且当有新的零件加入时，只需继承 Part 抽象类即可，现有的代码不需要做任何修改。但是，一旦需求发生了变化，如在计算零件价格和的时候还需要考虑市场价格波动情况，这时需要修改代码，如图中的注释部分，但这将导致该设计不再符合 OCP。

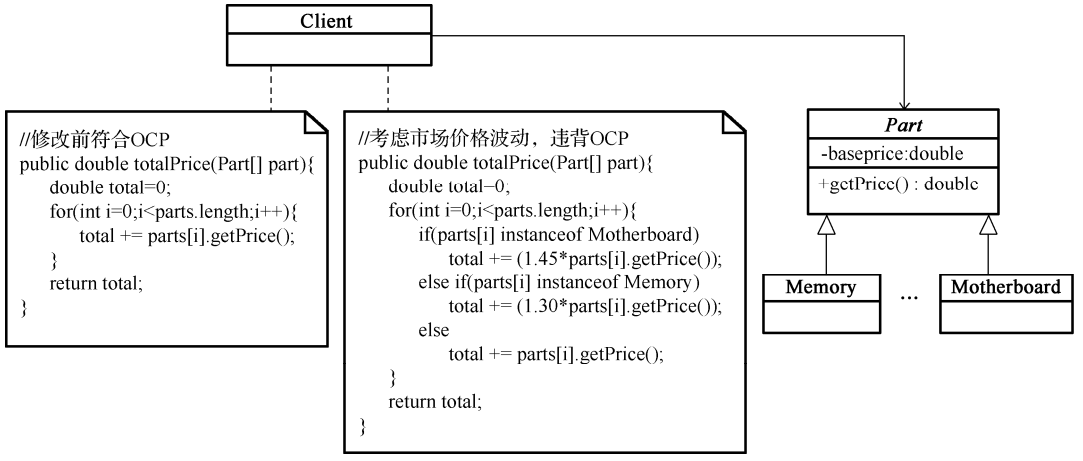


图 8.9　OCP 示例

作为对以上设计的一种改进，我们需要将市场价格波动的处理融入业务逻辑中，引入折扣(discount)属性。但这一属性放在什么地方？显然，它不是 Part 类的自然属性，因为这一属性是与价格策略相关的，为此我们引入了 PricePolicy 类，与 Part 类呈关联关系，这样做的好处是，Part 对象与 PricePolicy 对象的关系是动态的，可以在程序运行时(Runtime)动态改变。改进后的 OCP 设计如图 8.10 所示。

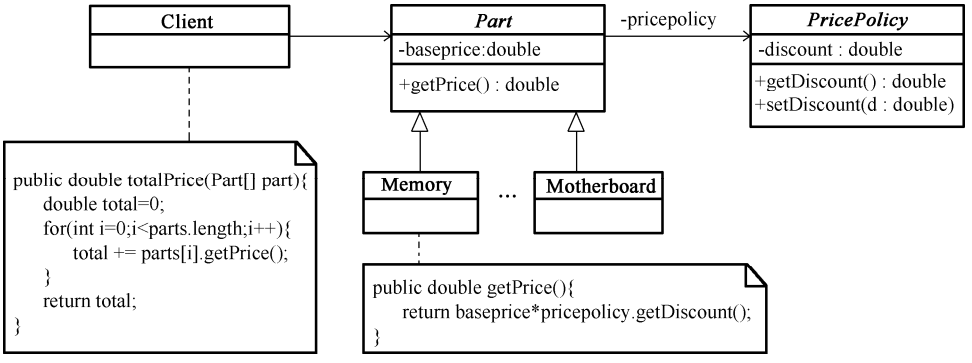


图 8.10　改进后的 OCP 示例

OCP 是面向对象设计的真正核心，符合该原则意味着最高等级的复用性和可维护性。

8.3.4 Liskov 替换原则

Liskov 替换原则(Liskov Substitution Principle，LSP)是指任何出现父类的地方都应该能使用子类对其进行无条件的替换，即当使用子类对其父类进行替换时，该组件仍像替换前一样正常工作。LSP 要求对象间的继承关系既与静态属性相关，又与动态行为相关。

对于行为的限制，通常要求父类在使用前和使用后都要具备必要的条件——前置条件和后置条件。当子类替换父类后，要求不能违反父类中的前置条件和后置条件，即一个子类不得具有比父类更多的限制。这是因为对于父类，某些使用是合法的，但是会违背子类的其中一个额外限制，从而违背 LSP。

一个简单和安全的做法是，不要将父类中子类不需要的函数暴露给子类，以减少违反 LSP 的可能性。代码 8.5 在行为上违反了 LSP。

代码 8.5　违反 LSP 的实现


```
//一个长方形类
public class Rectangle {
    private double width;
    private double height;
    public Rectangle (double w, double h) {
        width = w;
        height = h;
    }
    //这里省略了 getter 和 setter 方法
    public double area() { return width*height;}
}
//一个正方形类
public class Square extends Rectangle {
    public Square(double s) {super(s,s);}
    public void setWidth(double w) {
        super.setWidth(w);
        super.setHeight(w);
    }
    public void setHeight(double h) {
        super.setHeight(h);
        super.setWidth(h);
    }
}
public class TestRectangle {
    //定义一个使用基类 Rectangle 的方法
    public static void testLSP(Rectangle r) {
        r.setWidth(4.0);
        r.setHeight(5.0);
```

```
        System.out.println("宽度为 4.0，高度为 5.0，面积：" + r.area());
        if (r.area()==20.0)
            System.out.println("计算没问题！\n");
        else
            System.out.println("计算错误！\n");
    }
    public static void main(String args[]) {
        Rectangle r = new Rectangle(1.0,1.0);
        Square s = new Square(1.0);
        //以下函数应该对长方形和正方形都是等效的
        testLSP(r);
        testLSP(s);
    }
}
```

8.3.5 单一职责原则

单一职责原则(Single Responsibility Principle，SRP)中的职责，可理解为功能，就是设计的类功能应该只有一个，而不应该为两个或多个。这里，职责是引起“变化”的原因：一个类中有两个以上的变化方向(维度)，会产生过多的变化点，不同维度上的变化点组合，会导致设计上的臃肿和不合理，如图 8.11 所示。

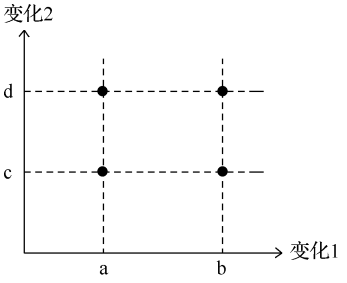


图 8.11　变化维度与变化点

图 8.12 为一个 SRP 的示例。设备 Modem 的设计功能存在两个变化方向：数据线路(DataChannel)和连接方式(Connection)。为了应对可能的需求变化，需要在子类中产生 4 个不同的类型，分别是 PxWaveModem、PyWaveModem、PxLineModem 和 PyLineModem。从设计的角度分析，这样的设计结果是我们不愿看到的，原因很简单：如果有 *n* 个变化方向，每个变化方向上有 *m* 个变化点呢？

如何应对这样的状况？答案就是拆分变化！将 Modem 拆分成两个抽象类：Connection 类和 DataChannel 类。需要说明的是，拆分后设计并未结束，两个拆分的抽象类肯定不是也不应该独立存在，读者可以思考一下如何进行下一步的设计[①]。

SRP 要求的条件比较苛刻，一个类真的要做到 SRP，只能有一个功能而一点儿其他功能

① 可以参照设计模式之“桥模式”。

也不能有？答案同样是否定的。多个功能在一个类中可以同时存在，但有一个前提：是否能够成为变化的维度。若具有单独的变化维度和变化点，则应按照 SRP 进行类职责的拆分，否则可以保留功能共存[①]。

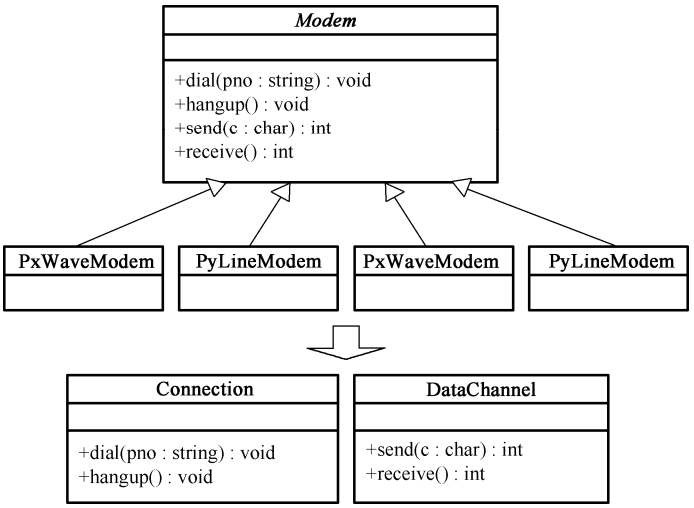


图 8.12　SRP 的示例

8.3.5　合成/聚合复用原则

合成/聚合复用原则(Composite/Aggregate Reuse Principle，CARP)中的合成与聚合是两种特殊的关联关系，是以委托方式实现对象之间功能的复用(重用)，除此之外，另外一种面向对象特有的重用方式是继承。委托重用与继承重用是两种本质上不同的重用方式，委托重用追求的是对象间的独立性即低耦合，而继承重用追求的是继承对象间的高耦合。

合成/聚合复用原则是指应尽量使用合成/聚合形式的委托重用，尽量不使用继承重用。具体地，在一个新的对象里面使用一些已有的对象，使之成为新对象功能的一部分(关联)：新的对象通过向这些对象的委托达到重用已有对象的目的。

图 8.13 给出一个违反了委托重用优先的例子。在这个例子中，乘客 Passenger 和代理 Agent 分别是 Person 的子类，而且利用多重继承，我们可以得到具有多重身份的特殊人——AgentPassenger，这一切看上去似乎没什么问题，但仔细分析一下，就会发现其中存在着诸多问题：

(1)多重继承中类型的确定是静态的，即人的身份一经确认，无法在运行时动态更改，而 Passenger 的身份本身就是动态的。

(2)多重继承需要生成大量的底层类，以适应具有不同身份组合的人群，如同时具有学生、代理、乘客、实习者等身份的人。

这些问题产生的原因就是 Person 类和 PersonRole 类其实是两个耦合性很低的实体，把它们放在一起，违反了事物的本质特性。一个改进的做法如图 8.14 所示，利用 CARP 降低两者之间的耦合度，从而将两者的真实特性表现出来。因此，优先使用关联的聚合或组合可获得重用性与简单性更佳的设计。另外，配合使用继承，可以扩展可用的组合类集，加大重用的范围。

① 具体请参照设计模式之“装饰模式”。

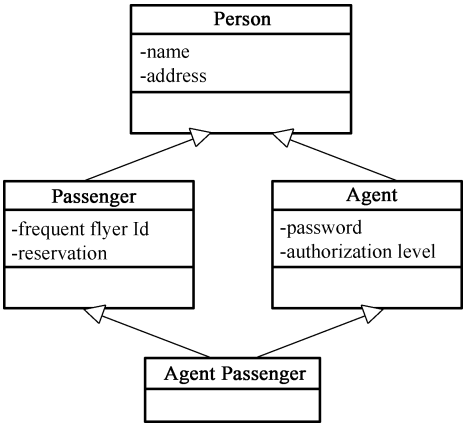


图 8.13　违反 CARP 的示例

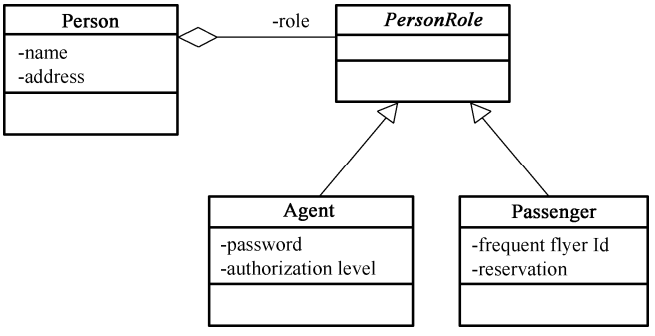


图 8.14　遵守 CARP 的改进设计

当然，在条件满足时，也推荐使用继承。例如：

(1) 子类表达了“是一个父类的特殊类型”，而非“是一个由父类所扮演的角色”；

(2) 子类的一个实例永远不需要转换为其他类的一个对象；

(3) 子类对其父类的职责进行扩展，而非重写或废除，因为这会增加违反 LSP 的可能性。

除前面提到的几个基本的设计原则外，还有其他一些原则同样能够实现好的设计，如 GRASP 模式(General Responsibility Assignment Software Patterns)。这里不再赘述，读者可自行查阅相关资料理解。

8.4　设计模式

设计模式提供了相似的程序设计任务中经常出现的相同问题的解决方案。基于工程化的方法，能够保证这些相似任务被识别出来，然后通过对设计模式进行较小的调整，快速产生问题的有效解决方案。设计模式提供了一种在设计层次上的重用机制，其特点是对解决方案的进一步抽象，一方面，模式针对的是抽象的设计思想的重用，而非代码的物理重用；另一方面，模式并不对具体问题提供完整的解决方案，而是提供方案的一种结构。

四人组(Gang of Four, GoF)的一本书 *Design Patterns: Elements of Reusable Object-Oriented*

Software 将 23 种设计模式总结为三类，分别是创建模式、结构模式和行为模式，如表 8.1 所示。下面分别就三类模式中有代表性的若干模式进行具体介绍，分别是创建模式中的抽象工厂(Abstract Factory)模式和单例(Singleton)模式，结构模式中的适配器(Adapter)模式、桥(Bridge)模式、装饰(Decorator)模式、门面(Facade)模式和代理(Proxy)模式，行为模式中的观察者(Observer)模式、策略(Strategy)模式和状态(State)模式。它们都是在工程实践中经常使用的模式，对设计优化也有重要的指导性。

表 8.1　设计模式一览

		目　的		
		创建模式	结构模式	行为模式
范围	类	Factory Method	Adapter	Interpreter
		—	—	Template Method
	对象	Abstract Factory	Adapter	Command
		Builder	Bridge	Observer
		Prototype	Composite	Visitor
		Singleton	Decorator	Memento
		—	Facade	Strategy
		—	Proxy	Mediator
		—	Flyweight	State
		—	—	Iterator
		—	—	Chain of Responsibility

8.4.1　抽象工厂模式

图 8.15 给出了抽象工厂模式的结构。抽象工厂模式的主要作用是解耦客户类(Client)在创建产品类(Product)时引入的耦合，如在对具体对象创建时使用的 new 操作，需要指定 Product 类的名字，这样就在 Client 类和 Product 类之间引入了依赖关系，而这种依赖关系按照面向接口编程等原则应该进行优化处理。可是，产品的创建必须使用具体类的名字，如何让 Client 类中不直接出现 Product 类的信息呢？采用的方法就是将产品的创建过程从 Client 类中分离，通过使用一个类似系统服务的工厂类来解决这个问题。工厂类提供了一个创建一系列相关或相互依赖对象的接口，Client 类无须指定它们需要的具体产品类。

在类图中，Client 类对 Product 类没有直接的依赖关系，这是由于通过抽象工厂类(AbstractFactory)进行了分离。另外，每个 AbstractFactory 类提供了对不同系列产品的创建。图 8.16 给出了一个抽象工厂模式的具体应用，其中主要关注棋类游戏的包装类 PackBox，该包装类中又含有棋盘和棋子两种产品。对于不同的棋类游戏(如中国象棋、跳棋或围棋等)，包装应具有不同的大小和体积。为了减少 PackBox 对象与这些棋类对象间的耦合程度，设计中使用了棋盘接口(ChessBoard)和棋子接口(ChessPiece)，同时利用一个抽象的棋类工厂类(ChessFactory)来消除对这些棋类对象创建的耦合。PackBox 对象的部分代码 8.6 如下：

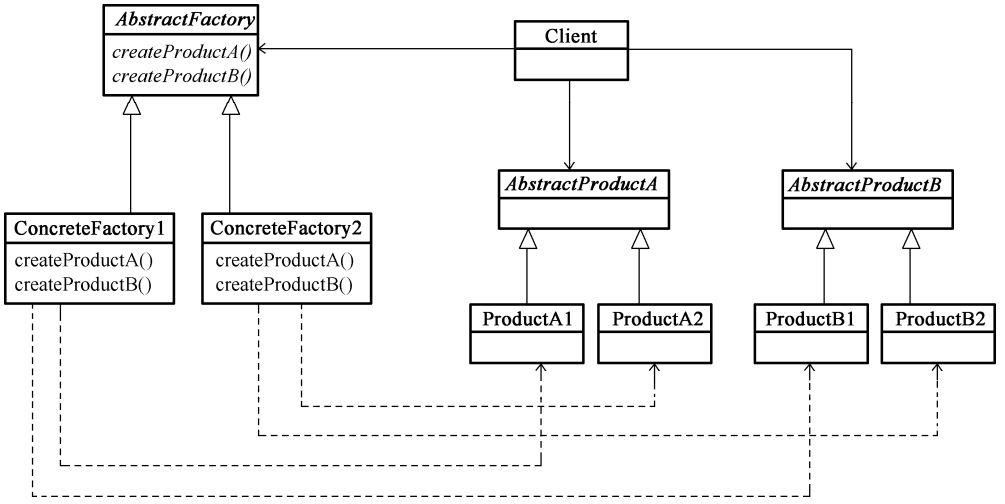


图 8.15　抽象工厂模式的结构

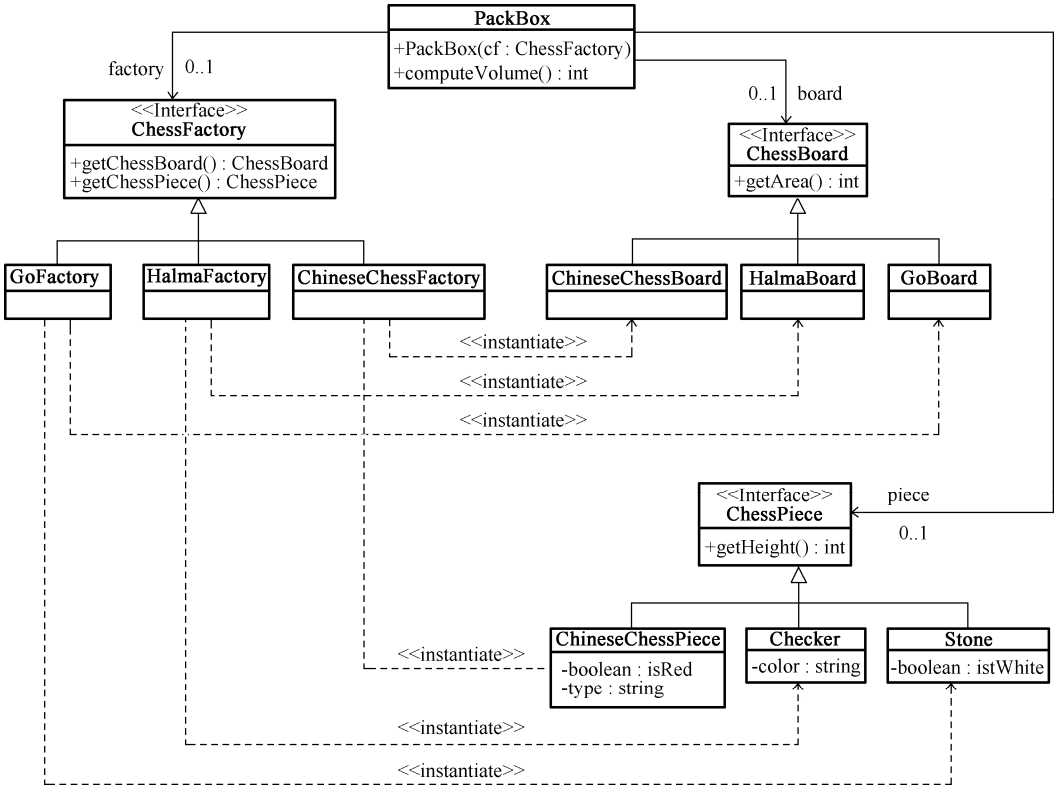


图 8.16　抽象工厂模式的示例

代码 8.6　PackBox 类的实现

```
public class PackBox{
    private ChessFactory factory=null;
    private ChessBoard board=null;
    private ChessPiece piece=null;
```

```
    public PackBox(ChessFactory cf){
        factory=cf;
        board=factory.getChessBoard();
        piece=factory.getChessPiece();
    }
    //......
    public int computeVolume(){
        return board.getArea()*piece.getHeight();
    }
    //......
    public static void main(String[] s){
        PackBox pb = new PackBox(new ChineseChessFactory());
        System.out.println("体积: " + pb.computeVolume());
    }
}
```

PackBox 类会对不同的棋类游戏返回不同的体积，这是根据传入的棋类工厂类决定的，ChessFactory 是抽象工厂类，传入的应为一个具体工厂类，如 GoFactory 类、HalmaFactory 类、ChineseChessFactory 类。ChineseChessFactory 类的实现如代码 8.7 如示，对 PackBox 类的使用可参照代码 8.6 中的 main() 函数。

代码 8.7　ChineseChessFactory 类的实现

```
public class ChineseChessFactory implements ChessFactory{
    ChineseChessBoard board;
    ChineseChessPiece piece;
    public ChineseChessFactory(){
        board=new ChineseChessBoard();
        piece=new ChineseChessPiece();
    }
    @Override
    public ChessBoard getChessBoard(){
        return board;
    }
    @Override
    public ChessPiece getChessPiece(){
        return piece;
    }
}
```

通过抽象工厂模式的使用，使得添加新的棋类游戏更加容易，只从棋盘类和棋子类继承并进行实现即可；但是，对于新种类的游戏，则需要完全重建一套工厂类和产品体系。

8.4.2　单例模式

前面章节中已介绍了管理类或控制类的概念，它们的使用主要是为了方便对相关的实体类进行管理和维护，因此它们在系统中通常只存在一个实例。类似地，根据业务的需要，在系统中可能还存在一些这样的对象实例，它们只存在单一的实例并且不允许生成其他副本，如银行系统中每个账户类的实例。

单例模式保证了一个类仅有一个实例，并提供一个访问它的全局访问点。单例模式要求：

(1)类的所有构造方法都为私有的，防止被外部创建；

(2)提供一个公有的方法获取该类的实例；

(3)类中的实例变量为私有或受保护的。

代码 8.8 展示了一个单例模式的类，并在代码 8.9 中对它进行了使用和测试。

代码 8.8　单例模式的类

```
package Singleton;
public class Singleton{
    private int x=0;
    private int y=0;
    private static Singleton pt = null;
    private Singleton(int x, int y){
        this.x=x;
        this.y=y;
    }
    public static Singleton getPoint(){
        if (pt == null)
            pt = new Singleton(5,39);
        return pt;
    }
    @Qverride
    public Singleton clone(){ //不允许自我复制
        return this;
    }
    public void output(){
        System.out.print("["+x+", "+y+"]");
    }
    public void move(int dx, int dy){
        x+=dx;
        y+=dy;
    }
}
```

代码 8.9　单例模式的使用和测试

```
package Singleton;
public class Main {
    public static void main(String[] s){
        Singleton pl=Singleton.getPoint();
        Singleton p2=Singleton.getPoint();
        //Singleton sing=new Singleton(); 该使用被禁用
        p1.output(); p2.output();
        if(pl == p2)
            System.out.println("\n same");
        pl.move(3,5);
        p1.output(); p2.output();
```

```
            Singleton p3 = pl.clone();
            if(p2 == p3)
                System.out.println("\n same");
        }
    }
```

请读者验证以上代码的输出，并体会单例模式的作用。

8.4.3 适配器模式

适配器模式把一个类的接口变换成该类期待的另一种接口，从而使原本因接口原因不匹配而无法一起工作的两个类能够一起工作。适配器模式一般有两种工作方式：一种是委托的方式；另一种是继承(接口实现)的方式。无论哪种方式，适配器都可以充当被适配对象参与类的交互，并可以对基本的适配功能做进一步的扩展，当然，这个功能扩展的设计要符合单一职责原则，或者可以应用“装饰模式”。

适配器模式的结构如图 8.17 所示。上图是通过继承的方式，而下图是通过委托的方式，其详细的代码注释和说明如代码 8.10 所示。

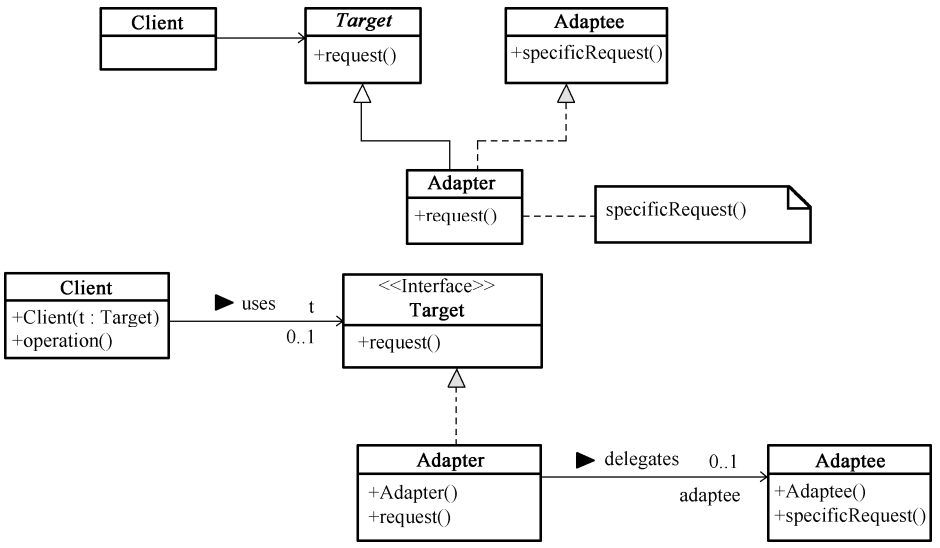


图 8.17　适配器模式的结构

代码 8.10　委托的方式的适配器实现

```
//Client.java
public class Client{
    private Target t=null;
    public Client(Target t){
        this.t=t;
    }
    public void operation(){
        t.request();
    }
```

```
    //...
    public static void main(String[ ] s){
        Client c = new Client(new Adapter());
        c.operation();
    }
}
//Adapter.java
public class Adapter implements Target{
    private Adaptee adaptee=null;
    public Adapte(){
        adaptee = new Adapter();
    }
    //...
    @Override
    public void request(){
        return adaptee.specificRequest();
    }
}
```


8.4.4 桥模式

桥模式的主要思想是将抽象部分与实现部分(行为)分离，使它们都可以独立地变化。本质上，它主要是用来解决两方面的变化问题：抽象与实现(行为)，即一个类中多个方向的变化问题。设计原则中的单一职责原则已经告诉我们，类要保持单个维度的变化。如果软件系统中某个类存在两个独立变化的维度，通过桥模式可以将这两个维度分离出来并使用关联的方式进行组织，使两者独立扩展。

桥模式用抽象关联取代了传统的多层继承，将类之间的静态继承关系转换为动态的对象组合关系，使系统更加灵活，并易于扩展，同时有效控制了系统中类的个数。桥模式的结构如图 8.18 所示。

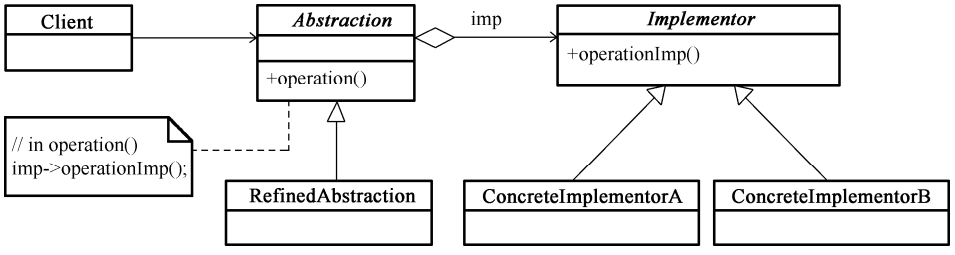


图 8.18　桥模式的结构

在桥模式中，体现了很多面向对象设计原则的思想，包括单一职责原则、开放封闭原则、合成/聚合复用原则、依赖倒置原则等。熟悉桥模式有助于深入理解这些设计原则，也有助于形成正确的设计思想和培养良好的设计风格。

使用桥模式时，首先应该识别出一个类所具有的两个独立变化的维度，将它们设计为两个独立的继承等级结构，为两个维度都提供抽象层，并建立抽象耦合。一般把业务方法和与之关系最密切的维度设计为“抽象类”层次结构(抽象部分)，而将另一个维度设计为“实现类”

层次结构(实现部分)。如对于咖啡饮料，由于口味是其固有的维度，因此可以设计一个抽象的咖啡类，在该类中声明并部分实现咖啡的业务方法，而将各种口味的咖啡作为其子类；咖啡饮料的容量是其另外的维度。该模式的实现方式如代码 8.11 所示。

代码 8.11　桥模式的实现

```
interface Implementor{
    public void operationImp();
}
abstract class Abstraction{
    protected Implementor imp;              //与实现类的关联(桥)
    public void setImp(Implementor imp){
        this.imp=imp;
    }
    public abstract void operation();//抽象业务
}
class RefinedAbstraction extends Abstraction{
    public void operation(){
        //业务代码
        impl.operationImp();                //具体实现
        //业务代码
    }
}
```


8.4.5　装饰模式

一个类，除了主体业务操作可能有些额外的操作，如加密、缓存、压缩等，这些只是辅助主体业务的附着，并不严格按照维度变化。装饰模式以对客户端透明的方式扩展对象的功能，是继承关系的一个替代方案，提供比继承更多的灵活性。它可以动态给一个对象增加功能，这些功能可以再动态撤销，因此可以增加由一些基本功能排列组合而产生的大量的功能。装饰模式中既有继承，又存在组合，实际上是将桥模式中的“抽象”和“实现”合二为一了，是桥模式的一种特殊形式。它的基本结构如图 8.19 所示。

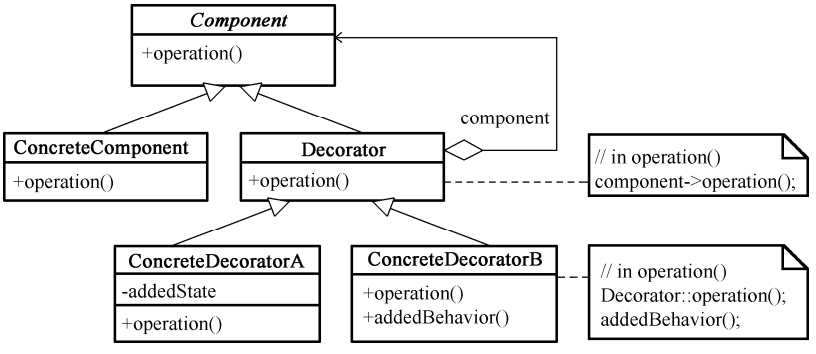


图 8.19　装饰模式的结构

装饰模式的核心在于抽象装饰类 Decorator 的设计，其实现如代码 8.12 所示。

代码 8.12　装饰模式的实现

```
class Decorator implements Component{
    private Component component; //关联原抽象业务
    public Decorator(Component component){
        this.component=component;
    }
}
class ConcreteDecorator extends Decorator{
    public ConcreteDecorator(Component component){
        super(component);
    }
    public void operation(){
        super.operation(); //调用原业务
        addedBehavior();   //扩展业务
    }
    public void addedBehavior(){
    //具体扩展业务的实现
    }
}
```


8.4.6　门面模式

门面模式要求外部与一个子系统的通信必须通过一个统一的门面对象进行。门面模式提供了一个高层次的接口，使子系统更易于使用。另外，每个子系统一般只要求具有一个门面类，而且此门面类只有一个实例。也就是说，它是一个单例模式。这时门面类的作用相当于前面介绍的适配器，负责对外部请求的转发，并且可以在此基础上进行功能的扩充，如对传递进来的参数的验证等。门面模式的结构如图 8.20 所示，一个系统可以有多个门面类。

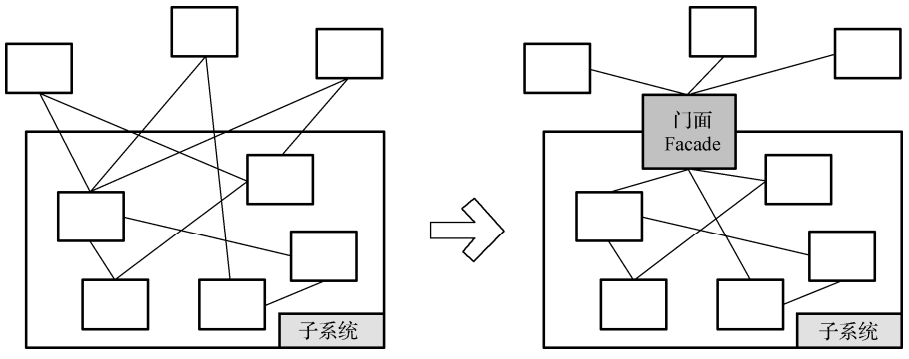


图 8.20　门面模式的结构

更常见的做法是，系统并不提供门面类，而是提供一个或多个门面接口。这对于系统内部的开发者来说是非常实用的，他们可以较为自由地实现内部功能，只要保证它们的行为具有门面接口中约定的行为即可。在大型的软件开发中，我们可以事先将各个子系统的外部行为确定下来，即门面接口，然后按照上述方法逐渐完善其内部设计和开发，这种方法称为“基于契约的设计(Design by Contract)”。

同时，门面类中集成了子系统中不同的内部功能，这是否违反了单一职责原则？门面类虽然具有多种功能，但它每次为外部提供服务时一般只涉及其中一类功能，几乎不会进行各种功能的联合使用，也就是这些功能大多是独立变化的，不会组合在一起形成多个变化点，因此本质上并不违反单一职责原则。由此可见，面向对象的设计不能生搬硬套，应视具体情况做具体分析。

8.4.7 代理模式

代理模式一般用于对有价值(稀缺)资源的管理，如数据库的连接等，目的是提高这些资源的利用率或者系统性能。它为这些资源对象提供一个代理对象，并由代理对象控制对资源对象的使用，起到中介的作用。代理对象的存在使实体类分辨不出代理对象与真实的资源对象。

代理模式的结构如图 8.21 所示，它的实现如代码 8.13 所示。

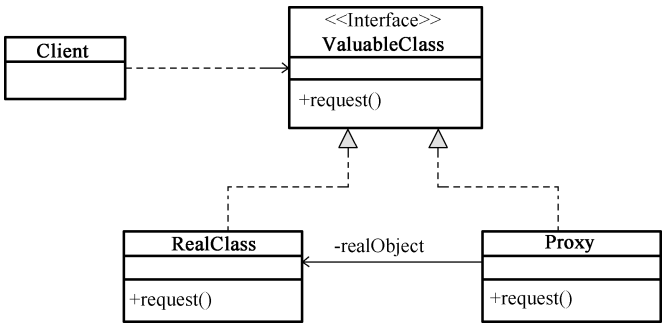


图 8.21　代理模式的结构

代码 8.13　代理模式的实现

```
public interface ValuableClass{
    public int request(String details);
}
public class RealClass implements ValuableClass{
    private Connection conn;
    public RealClass(String conndata){
        conn = new Connection(conndata);
    }
    public int request(String details){
        return conn.execute(details);
    }
}
public class Proxy implements ValuableClass{
    private static RealClass realObject;
    public Proxy(){
        if(realObject == null)
            realObject = new RealClass("ConnectionString");
    }
    public int request(String details){
```

```
            return realObject.request(details); //这里可以加入对资源使用的控制
        }
    }
    public class Client{
        public int proxyUse(String s){
            ValuableClass v = new Proxy();
            return v.request(s);
        }
        public static void main(String[] s){
            Clinet n = new Client();
            System.out.println(n.proxyUse("whatever"));
        }
    }
```

对于 RealClass 类，这里只是一个简单的示意，我们假定它含有某个对外部系统的连接，并通过该连接实现某些资源的请求。代理类 Proxy 中持有对资源类 ValuableClass 的引用，Proxy 类控制着对资源的具体使用方式和策略。代码中虽然没有复杂的资源控制，但也能够说明如何实现对 ValuableClass 类的请求进行控制和转发。客户类 Client 的代码实现对 Proxy 类的使用。

代理模式也可以并不知道真正的被代理对象，而仅仅持有一个被代理对象的接口，这时代理对象不能够创建被代理对象，被代理对象必须由系统的其他角色代为创建并传入，如“动态代理”机制。

8.4.8 观察者模式

观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象；当这个主题对象在状态上发生变化时，会通知所有观察者对象，使它们能够自动更新自己。MVC 结构在实现上就使用了观察者模式，其中的主题对象相当于 MVC 中的模型，观察者对象相当于 MVC 中的视图。

图 8.22 给出了观察者模式的结构，每个观察者(Observer)对象为了得到主题(Subject)对象的及时通知，需要事先在 Subject 对象中进行订阅，并且在不需要时进行退订。每个具体的 Observer 对象需要实现自己的更新方法 update()。

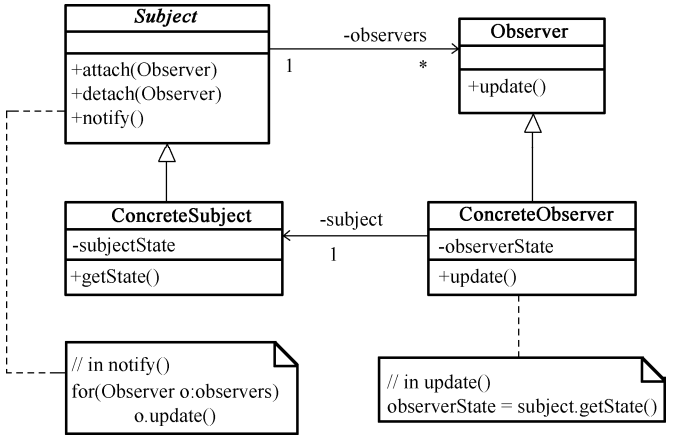


图 8.22　观察者模式的结构

8.4.9 策略模式

工程实践中经常会求解针对复杂问题的解决方案，而这些方案经常使用某种算法进行描述，而且同一问题在不同的情况下可能会采用不同的算法，如排序问题就有冒泡排序、快速排序、合并排序等算法，它们有着各自不同的特点，应用于不同的需求中。策略模式针对一组算法，将每一种算法封装到具有共同接口的独立的类中，从而使它们可以相互替换。

策略模式的好处是能够使算法可以在不影响客户端的情况下进行切换，而且将算法的行为和环境分开，环境负责维持和查询行为，各种算法在具体的策略类中提供。由于算法和环境是相互独立的，算法的增减、修改都不会影响到环境和客户端。图 8.23 给出了策略模式的结构，相关实现如代码 8.14 所示。

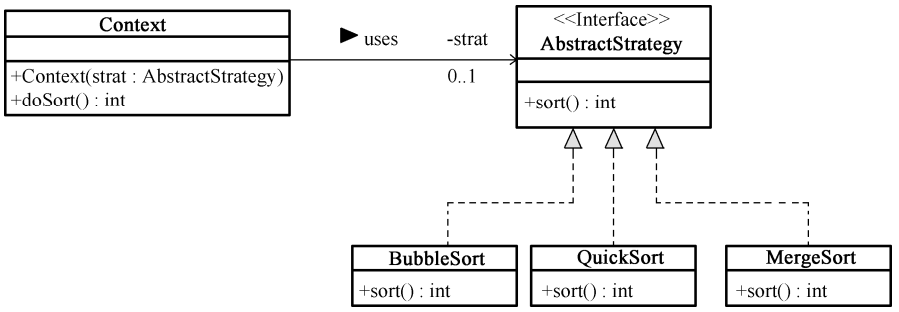


图 8.23　策略模式的结构

代码 8.14　策略模式的实现

```
//Context.java
public class Context{
    private AbstractStrategy strat = null;
    public Context(AbstractStrategy a){
        this.strat=a;
    }
    //...
    public int doSort(){
        return strat.sort();
    }
}
//Strategy.java
public class Strategy {
    public static void main(String[] args){
        Context c = new Context(new BubbleSort());
        c.doSort();
    }
}
```

8.4.10 状态模式

状态模式可以视为策略模式的一种应用，状态模式允许一个对象在其内部状态改变时改

变行为。状态模式把所研究对象的行为包装在不同的状态对象里，每个状态对象都属于抽象状态类的子类。

当系统的状态变化时，系统便改变所选的子类，从而对类在不同状态下的行为进行管理。

下面的示例是一个测量工作站的工作状态描述，图 8.24 上面的子图描述了该工作站的状态图，其中 x 为某测量值(如温度)。当 x 的值小于 42 时，为正常状态，否则会过渡到临界状态；在临界状态下如果 x 回到 22 及以下，该工作站又会返回到正常状态。图 8.24 下面的子图为对应的状态模式的应用结构，这里存在一个抽象类 State，其 setX()方法的主要作用是对状态变量 x 进行修改，另外两个具体状态 StateOK 类和 StateCritical 类分别对应正常状态和临界状态的实现。该系统的实现如代码 8.15 所示。

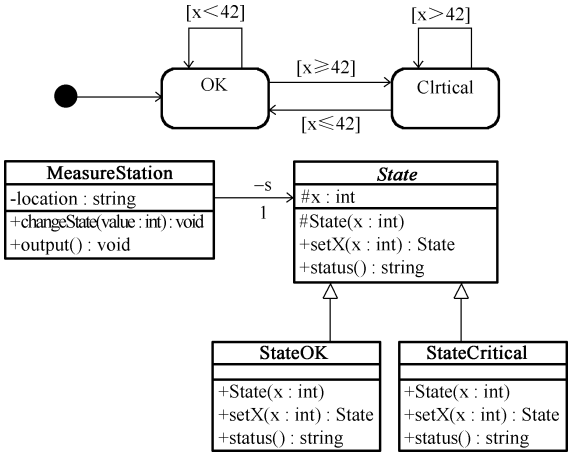


图 8.24　状态模式的示例

代码 8.15　状态模式的实现

```
//State.java
package State;
public abstract class State{
    protected int x;
    public abstract State setX(int x);
    public abstract String status();
    protected State(int x){
        this.x=x;
    }
}
//StateOK.java
package State;
public class StateOK extends State{
    public StateOK(int x){
        super(x);
    }
    @Override
    public State setX(int x){
        this.x=x;
        if(x>=42) return new StateCritical(x);
```

```
            return this;
        }
        @Override
        public String status(){
            return "ok";
        }
    }
    //StateCritical.java
    package State;
    public class StateCritical extends State{
        public StateCritical(int x){
            super(x);
        }
        @Override
        public State setX(int x){
            this.x=x;
            if(x<=22) return new StateOK(x);
        return this;
        }
        @Override
        public String status(){
            return "critical";
        }
    }
    //MeasureStation.java
    package State;
    public class MeasureStation{
        private String location = "DL";
        private State s = new StateOK(0);
        public void changeState (int value){
            s = s.setX(value);
        }
        public void output (){
            System.out.println(location + " State: " + s.status());
        }
    }
    //Main.java
    package State;
    public class Main {
        public static void main(String[] args) {
            MeasureStation m = new MeasureStation();
            int[] values={18,42,38,20,45};
            for(int i:values){
                m.changeState(i);
                m.output();
            }
        }
    }
```

在学习本章内容的过程中，建议读者不仅要从理论上理解每个模式的原理，为了加深印象，为每个模式对应写出一个小的示例性的代码也很有必要。

8.5 习题

1．图 8.25 所示的类图的设计中使用了观察者设计模式，每个券商(Broker)关注的是股票(Stock)的价格，它们可以将感兴趣的股票设定为自选股，并通过方法 newStock()购买指定的股票。

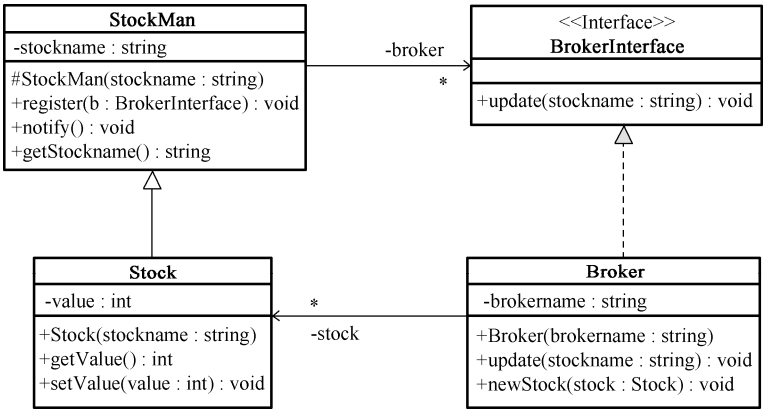


图 8.25　类图

如果股票的价格发生了变化，所有选定该股票为自选股的券商将会得到通知，包括该股票的名字，这里假定每个股票有唯一的名字。使用 Java 语言对该类图进行实现，要求券商得到股价改变的通知后，输出该券商的名字、股票的名字及股票的最新价格。可将该系统的界面设计为如图 8.26 所示的形式。



图 8.26　界面

2．图 8.27 是合成(Composite)模式的结构。合成模式把部分与整体的关系用树结构进行表示，使用户对单对象和组合对象的使用具有一致性。

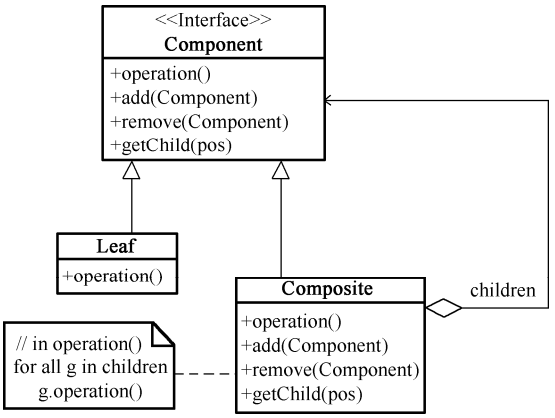


图 8.27　合成模式的结构

请给出一个合成模式的具体应用，并使用 Java 语言(或其他面向对象语言)进行实现。

3．熟悉和掌握表 8.1 中给出的其他设计模式。

第9章 实 现 技 术

从类图到代码的过渡在第 6 章中已经进行了说明，重点介绍了从类图到代码框架的产生过程。实现阶段的一个首要任务就是面向业务构造合适的代码架构，以使所有的用户需求能够在未来的系统中得到满足和体现。从设计到代码的转换过程不仅需要有算法过程的实现，同时还要考虑具体业务的约束条件。

本章不涉及具体业务的实现细节，而是重点关注那些与实现相关的关键技术及衍生内容。实现技术虽然与业务是独立的，但它们是业务实现的载体，合适的技术选择对业务的实现会起到有效的辅助和促进作用，反之则会拖累项目整体的进展，有时可能会导致整个项目的失败。

软件工程发展日新月异，无论是技术还是工具都存在着不同的流派，它们百花齐放，各有所长。本章不能涵盖所有当今流行的技术，也不去探讨孰优孰劣，而是着重讨论软件开发项目中常见的一些工程和技术问题，它们对软件工程的发展也有不同程度的影响。这些内容包括对数据管理方式的不同实现策略和方法、对数据进行持久化的方案、可扩展标记语言 XML 等。本章还将介绍领域特定语言(Domain Specific Language)、模型驱动架构(Model Driven Architecture)、重构(Refactoring)等技术内容。

9.1 非功能性需求的实现

非功能性需求对于项目的成功具有重要的作用。如果项目中存在非功能性需求的遗漏，将会使客户无法接受，项目也有可能就此夭折，因为这样的系统最终可能根本无法在合同约定的环境中运行。由于非功能性需求涉及的范围广且类型不尽相同，因此需要在设计和实现中根据不同的要求区别对待。下面举例说明几种非功能性需求及其对实现的影响。

1. 硬件方面的需求

项目在开始阶段一般通过原型系统的开发实现系统的模拟，用以获取用户需求，消除沟通上的壁垒，释放开发风险。原型系统一般不太注重性能上的表现，但性能表现一般是用户最看重的非功能性需求，而提高系统性能最直接的因素就是硬件资源。对于硬件资源的需求，要根据项目的实际情况进行选择，若系统经常处理单一的数据记录，通过图形界面对输入或输出进行简单的格式或合法性校验，则硬件的计算能力和基础构架就显得不太重要；但是在复杂计算的需求下，如对海量数据的分析或者对数据的实时处理，就需要在开发中更加重视性能上的表现，采用多种措施提升系统整体性能，如使用多台 Web 服务器或应用服务器进行负载分担、热点数据缓存、分离数据库的读写请求以减轻单台服务器的 I/O 负担等，这些措施将会影响系统的架构设计。很多企业采用类似图 9.1 所示的集群架构来提升系统整体性能。

另外，开发过程要关注系统运行时间和数据访问优化的重要性，尽量降低程序中资源浪费的可能性，否则开发的成本会显著提高。在实现方面，可以引入一些性能度量的工具，以对系统运行剖面(Profile)进行分析，如利用工具对语句的执行频率进行测量，并确定系统的瓶颈。

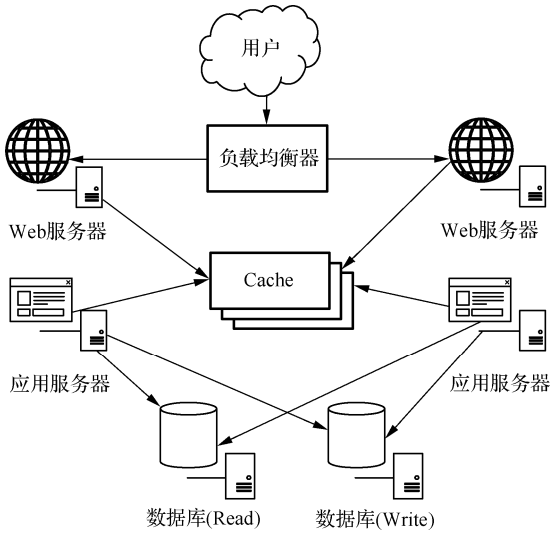


图 9.1　企业级集群架构

2. 质量方面的需求

某些应用可能关乎人员的财产和生命安全，这就要非常注意软件的质量，即系统的正确性。质量涉及整个软件开发过程，包括一系列质量保证相关活动的应用和组织，这将在第 11 章中具体介绍。

质量的需求在不同的应用领域对于系统的实现具有不同的影响。除了可测试性的需求，也可能对程序的结构或内部实现提出限制，如在一些要求比较严格的应用中不允许动态地对内存区域进行分配，这就限制了动态数据结构的使用，如链表或递归，在实现上就需要在开始阶段估算出每项功能需要的存储空间大小，从而估算出最大的内存要求，以避免存储空间的溢出。

3. 安全方面的需求

系统的安全方面存在两种不同的解释，一种特指授权安全性(Security)，另外一种是数据存储安全性(Safety)。如果需要处理个人隐私数据等安全方面的敏感信息，就需要特别关注授权安全性。当信息通过不安全的外部通道(如 Internet)进行传输时，尤其要注意数据的传输保护，如采用 https 访问协议。对于授权安全性的要求，实现上采取的方法可以借助软件或者硬件加密机制，但这些手段都会不同程度地降低程序的运行效率或者增加系统结构的复杂程度，如引入数字签名或者证书认证(CA)等机制。数据存储安全性则主要考虑系统从故障中恢复的能力，同样，在实现上需要进行额外的考虑，包括各种备份技术的应用，如数据备份、双机热备、RAID 等。

随着分布式计算的流行，区块链(Blockchain)技术发展迅速，它在分布式环境中通过构建超级账本来实现一种去中心化的安全保证，支持内容不可更改、共识信任和隐私保护等原则，是一种建立新型的分布式商业网络的新技术。另外，还存在其他一些分布式系统，如 Zookeeper 等，其借助一致性协议[①]，可以很好地解决安全方面的问题。

① 如 Paxos、Raft 等分布式一致性协议。

9.2 分布式技术

集中式的应用一般通过集中的程序流程进行控制，虽然涉及新对象的创建或者对各种方法的调用，但通常程序中都存在唯一的控制点，决定下一步的执行内容及对其他方法的调用。分布式系统中存在多个这样的控制点，因为有多个子程序需要同时工作。例如，分布式操作系统的进程，它们能够对不同的硬件单元进行控制和管理。系统将任务划分为多个进程分别处理，这些进程之间相互独立，较少进行信息的交换。

进程间信息交换的方式和规则是计算机领域中的一个较为复杂的问题，解决的方法与很多其他的条件要求密切相关。例如，在数据库领域中，使用事务的概念描述一组逻辑密切相关的操作，保证其执行不会对其他事务产生副作用。通过事务，能够保证银行账号间的转账操作在任何情况下的数据一致性，即扣款和入账的同时成功，或者失败后全部回到扣款前的状态，进而保证事务的特性之一——原子性[①]。

理论上，我们将这样的范围称为临界区，其限定了一个不能被其他进程中断操作的范围。例如，存在两个进程对全局变量 X 进行使用，每次使用对 X 增加 1，就会出现下面的情况：如果计算过程没有原子性，两个进程对 X 的访问实际上都要先获取 X 的值，并在本地副本中进行保存然后加 1，最后再回写全局的 X。由于对副本的操作在两个独立的进程中进行，因此会造成先写回的值被后写回的值覆盖，即变量 X 只被增加了一次，从而无法保证数据的一致性。

进程间的通信也存在多种不同的形式，一些形式也只能被某些编程语言支持。例如，一种简单的开发方式是通过所谓的远程过程调用(RMI)协议实现分布进程的通信，通过一个对开发者透明的控制进程实现对远程系统中方法的调用，此过程可以按照 8.4.7 中介绍的代理模式进行设计和实现。从开发的角度来看，这是一个同构的系统，由于存在网络间的调用，效率上会打一定程度的折扣。另外，在实现的层面上还要考虑远程对象调用的管理，这里存在两种方式：一种方式是每个对象只在其原始被创建的进程中进行管理，所以对此进程中该对象的方法调用都需要进行转发；另外一种方式是将对象及其副本在所有的网络节点上分布，这样对该对象的调用都是本地调用，副本间的同步由副本本身负责。

另外，进程间的通信还需要关注同步调用和异步调用。

同步调用是指消息发送者处于等待状态，直到接收者准备好提供服务，然后接收者执行需要的计算，发送者仍然等待，直到获得接收者返回的结果。异步调用中，发送者将请求信息发给接收者，然后自己不中断地继续后续工作，并在某个时刻(如空闲)检查请求的服务结果是否已经返回。对方的接收者接收到请求后会对信息进行处理，并将结果放置在一个缓存的空间进行管理。

同步调用具有的最大优点是所有的进程相互了解各自在通信过程中所处的状态，缺点是需要实现相对复杂的同步协议。由于发送方和接收方需要相互等待，所以会使两个进程在总体上的执行速度变慢。

① 事务具有 ACID 特性，即原子性、一致性、隔离性和持久性。

图 9.2 描述了一种典型的分布系统中存在的问题尤其是在同步调用的过程中更容易出现，即死锁的现象。图中，进程 A 请求 B 的服务，同时进程 B 也正在请求进程 A 的服务。在同步调用中，会出现两个进程相互等待的情况，这就导致了死锁的产生。

异步调用的执行速度通常比较快，因为发送方和接收方可以互相独立地工作。异步调用容易出的问题是缓冲区满了，这时整个系统运行变慢，信息可能丢失。另外一个问题如图 9.3 所示。从图中能够清晰看出，发送方和接收方及它们的进程彼此调用的顺序。进程 B 假设进程 C 的调用先于进程 A 的调用，但实际上并不一定是这样的。进程 A 假设其调用 C 的方法在来自 C 的调用请求被处理之前完成，而在进程 C 假设其调用 A 的方法在来自 A 的调用请求被处理之前完成，我们将两种执行流程称为进程竞争。

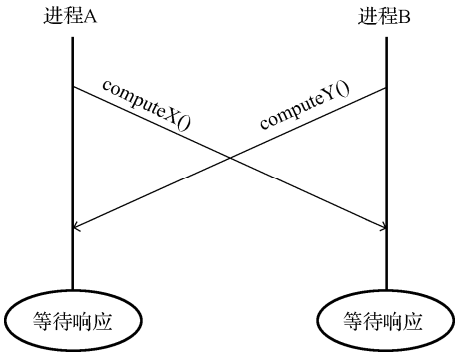


图 9.2　进程死锁

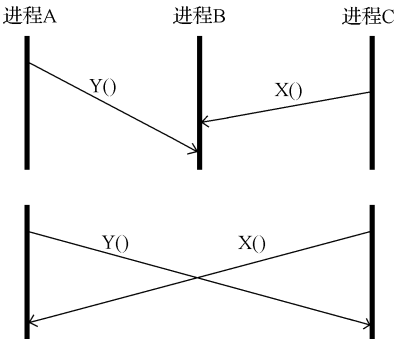


图 9.3　进程竞争

除了死锁问题，在分布式系统中可能还存在活锁问题。活锁问题是指两个处于活动状态的进程都在执行新的指令并更新自己的状态，但系统总体上的状态和进度并没有前进。这好比两个人在一个狭窄的走廊里相遇，两个人都同时礼貌地回避让对方先行，但每个人仍处于原地，两人步调一致地反复进行下去，谁都没能前行。这就是活锁的例子，又称饿死(Starvation)。

相互竞争的进程如果都满足对某一关键资源的占有条件，采用排队机制的先来先服务原则相对最公平，但有时会导致某一进程占用一种资源的时间过长，致使其他资源长期闲置的现象。适当地将资源释放并分配给那些等待时间长、占时少，又重要的进程更为公平。因此，对于进程的执行，一个更具体的问题是进程的调度问题。在操作系统中，我们一般采用进程优先级的方式来解决此问题，即具有高优先级的进程具有被选中并执行的优势，而且优先级一般是动态设定的，随着进程等待时间的延长，进程的优先级也要进行一定的调整。优先级设置不当，就会造成某些进程永远处于阻塞状态而产生死等。死等是不公平调度引起的。解决的办法是改变某些进程优先级的设定算法，在公平性和合理性上进行折中。

除了对分布系统中进程执行的细节控制，还需要明确哪些任务由哪些进程负责完成，这对于在不同计算机上分布的多进程的系统或软件来说尤其重要。一个设计不佳的进程方案中，可能会存在某个中心进程，当某些进程依赖此中心进程的计算结果时，就可能使这些进程无法继续执行，即并发性不强。

例如，在传统的客户机/服务器(C/S)结构中，其进程分布的结构一般是一个中央协调者(服务器)与多个在不同计算机上分布的程序(客户机)进行通信。中央服务器通常要具有较强的计

算能力，以保证不会给客户机带来处理上的瓶颈；同时，还要对业务逻辑功能部署的位置(如哪些放置在客户机，哪些放置在服务器)做好规划。

图 9.4 中描述了两种可能的选择，在左边的“瘦”方案中，所有的复杂业务计算都在服务器上实现，客户机只负责数据的输入和输出，即与用户的交互。这种方式的优点是客户机的功能负载不大，因为只有简单的输入和输出处理；开发者对服务器端的应用具有最大控制权，能够确定哪些功能以何种方式被执行；客户机可以很容易地进行替换，因为它们不负责主要的业务功能实现。这种部署的缺点是会带来服务器和网络的高负载，因为每一次客户机与服务器的交互，都需要服务器机的计算和控制。

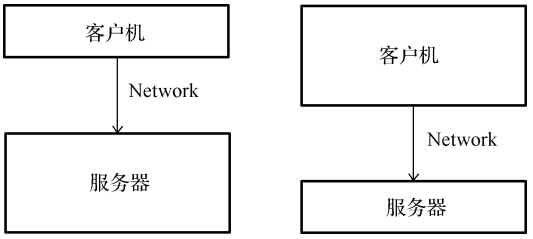


图 9.4　客户机/服务器结构(瘦和胖)

潜在的网络负载是产生另外一种方案“胖”的主要因素。客户机的功能除了基本的输入和输出，还将一些业务逻辑功能放置在本地，这减轻了服务器的负担，不再需要服务器具有很强的计算能力。由于业务功能的本地化，客户机可首先尝试在本地进行计算，只有需要时，才与服务器进行通信，这样网络负载也得以降低。

典型的层次架构及由此衍生出的多层架构，在提升系统的可维护性的同时，也为系统的分布提供了更好的选择，尤其是在 Internet 应用中，这样的架构有着广泛的应用。

9.3　可扩展标记语言 XML

本节介绍 XML 技术的规范及应用。XML 是由 W3C 委员会定义的一种标准化语言，用来描述数据模型和数据。标准化使得不同领域的数据按照统一的数据交换格式进行定义，可以将不同厂商开发的不同工具基于共同的数据模型更加容易地进行集成，基本原理是将每个领域的数据使用某种 XML 数据格式进行描述，不同领域的数据可自由进行 XML 格式的转换。

代码 9.1 描述了一个 XML 文档的层次结构，起始处是 XML 版本号及语言编码的说明。每个 XML 元素包含一个开始标签和一个对应的结束标签，其中可以嵌入其他 XML 元素和文本内容。在最外层的元素中存在唯一的元素，即根节点。示例中根节点<project>之内嵌入了<projectname>等多个节点，节点<projectleader>又由三个子节点构成。XML 结构的确切描述是由两种形式进行限定的：DTD(Data Type Definition)或 XSD(XML Schema Definition)。如果 XML 文档对标准的 DTD 或 XSD 进行了说明并进行了约束，则称该文档为 well formed，它们指定了该 XML 文档的有效性，详细规定了文档中元素内容和层次关系。

代码 9.1 中也给出了对 XML 元素属性的描述，第 3 行中属性值的内容要求放置在双引号中。如果一个 XML 元素只有属性而没有任何子元素，则可以使用简化描述方式，省略其结束标签，如第 5 行所示。XML 中的注释与 HTML 中的写法一样，如第 2 行。

代码 9.1　XML 示例

```
<? xml version="1.0" encoding="utf-8" ?>
<!-- this is a project -->
<project department="Development" contract="Fixedprice" >
   <projectname>Storage Module</projectname>
   <customer bankno="75566445" accountno="35634534"/>
      <projectleader cost="1000" >
         <empno>49</empno>
         <name>Udo Kelter</name>
         <deptno>50</deptno>
      </projectleader>
</project>
```

XML 文档的存储和处理，有很多可用的软件包或系统，使得每种编程语言都具有处理 XML 文档的能力。XML 的处理方式一般有两种：文档对象模型(DOM)或用于 XML 的简单 API(SAX)。当 XML 比较复杂或者当需要随机处理文档中的数据时，DOM 是复杂对象处理的首选；SAX 则是以流的方式从文档的开始处通过每一个节点进行移动，以定位一个特定的节点。

DOM 为载入内存的文档节点建立类型描述，表现为按照 XML 文档的层次组织的树结构。如果 XML 很冗长，DOM 就会显示出无法控制的扩展，占用巨大的内存资源。而 SAX 文档不需要一次性解析，它也不会有大量数据常驻内存中。因此，SAX 是一种“更轻巧的”XML 处理技术。如果需要处理较为复杂的操作，如高级 XSLT 转换或者 XPath 过滤，建议使用 DOM 方式；如果需要建立或者更改 XML 文档，也建议选择 DOM 方式。相反，可以使用 SAX 来查询或者遍历 XML 文档，SAX 可以快速扫描一个大型的 XML 文档，当它找到内容时，会通过回调进行处理。

XML 在实际工程中的应用主要有两个方面：一是用于轻量级的数据存储，尤其是配置信息等，不需要将其存储于大型的后台数据库[①]中；二是用于数据交换，这得益于它的平台和语言无关性。当然，XML 在实际应用中也存在一些问题，如 XML 文件格式通常比较复杂，传输占带宽，无论是服务器还是客户机都需要花费大量代码来解析 XML，导致代码变得相对复杂且不易维护。因此，工程实践中还会见到另外一种轻量级的数据交换格式 JSON(JavaScript Object Notation)，它的数据格式比较简单、易于读写、格式可以压缩、传输占用带宽小。两者之间的选择要根据实际的需求来定，例如，在一些 Web 应用中，JSON 更方便，但在 Web Service 中，XML 的支持要更成熟。

9.4　程序轮子

有经验的工程师善于在前人工作的基础上进行开发，而不是每次都重新探索已做的工作。“不要重复造轮子(Stop Trying to Reinvent the Wheel)”说的就是这个道理，这对开发者也是一条很有意义的座右铭。最基本的轮子是指，对于大多数经常出现的问题，将它们优质的解决

① XML 也提供了类似 SQL 的查询语言 XPath 和 XQuery。

方法通过函数库或者类库的形式提取出来，作为一种公共的共享资源。程序轮子还可以是组件、框架等资源，这意味着应该多利用现有解决方案的知识积累，尽量缩短复杂而冗长的开发过程。

C++的标准模板库(STL)和 Java 的类库是两个较为常用的公共程序库，两个类库都提供给使用者某些可以直接使用的动态数据类型，如链表和集合，免去了独自实现的麻烦。当然，在实现上，两者都具有各自的特点，对于链表(List)类型，Java 的 List 能够直接以下标的形式访问第 i 个元素，而在 C++中则不可以。Java 类库中存在大量的程序包，其中含有较为丰富和强大的功能供开发者使用，如网络、语音及三维建模等。除此之外，还有大量的第三方开发包，这些轮子有的是开源免费的，有的是商业收费的，有的功能简单易用，有的功能强大，因此不要重复造轮子其实并不很容易做到。

(1)如何发现合适的轮子，特定的问题是否存在一劳永逸的轮子？这对信息检索能力有非常高的要求。

(2)找到了轮子，但这个轮子是否好用。还需要时间来论证。判断轮子的质量及易用性，需要大量项目经验的积累。

(3)如何有效地将轮子整合到自己的项目中？这需要对这个轮子有深入了解。

也可能会同时找到多个轮子，这要根据项目的实际情况进行取舍：有些轮子侧重大而全，希望解决大部分问题，但是细节上可能处理得不够细致；有些轮子小而精，专注解决一个问题，但是不具备好的通用性。有价值的轮子应该具有的特征包括：

(1)成熟稳定，代码规范，接口友好，功能符合期望；

(2)有相关支持的保证，如文档健全，最好要有实际用例；

(3)社区相对活跃，用户众多，维护频繁，缺陷处理及时；

(4)松耦合，定制容易；

(5)价格合适，有适合的使用许可，最好是开源的。

综上所述，寻找轮子的任务要由经验丰富，信息检索能力强的人员负责，最好是项目的构架人员。开发者平时要多积累，多关注 GitHub、SourceForge、Google Code 等项目托管网站上的优秀项目。

9.4.1 组件

组件是一种程序轮子，可以把它理解为一种特殊的对象。组件是对数据和方法的简单封装。使用组件可以实现拖放式编程、快速的属性处理及真正的面向对象设计。组件是对类库思想的进一步提升，不是仅提供单一类的功能，而是将某个子应用封装，提供使用。组件可以对接口进行实现，从而提供实现了这些接口的一类对象。使用现成的组件开发应用程序时，组件一般可以工作在设计时态和运行时态两种模式下。

在设计时态下，组件显示在窗体编辑器下的一个窗体中，设计时态下组件的方法不能被调用，组件不能与最终用户直接进行交互操作，也不需要实现全部功能。在运行状态下，组件工作在一个已经实际运行的应用程序中，组件必须能够正确地将自身表示出来，它需要对方法的调用进行处理，并实现与其他组件之间有效的协同工作。

创建组件就是自行设计制作出新的组件，设计组件是一项繁重的工作。自行开发组件与使用组件进行可视化程序开发存在着极大的不同，开发要求开发者熟知原有的类库结构，精

通面向对象程序设计。对于组件的开发者，组件是纯粹的代码，因为组件本身的开发一般不是可视化的开发过程，而是使用编程工具编码的工作。实际上，创建新组件使我们回到传统开发工具的时代，虽然这是一个复杂的过程，但会带来一劳永逸的效果。

组件的使用是一个相对轻松的过程，除了可以使用组件提供的大量功能，还可以对它们进行定制。组件的定制通常可以通过配置文件的形式进行，也可以通过交互的方式对组件的属性进行配置。例如，使用 Java 中 AWT 和 Swing 类来描述用户界面的各种构成，如按钮、文本框等。在 GUI 集成开发环境中，这些组件都是可以配置的，图 9.5 显示了开发环境 Netbeans 中的一个 GUI 设计界面，右下角部分是按钮属性的配置列表。



图 9.5　Netbeans 中的组件

组件本身是个较为宽泛的概念，具体的含义与开发的业务领域相关。组件与系统及组件之间的通信一般是按照观察者模式进行的。在分布式系统中，除了组件，还可以通过其他方式对外界提供服务，如 Web Service 等。

Java 中的组件支持机制是 Java Bean。Java Bean 是一种特殊的类，在组织上要遵照一定的设计规则，类似的还有微软的 ActiveX 等。Java Bean 必须存在一个默认的构造函数，即无参数的构造函数。对于每个含有的实例变量，必须存在其简单的 get 和与其类型相符的 set 方法。该类必须实现接口 Serializable，以具备序列化的能力。按照这些要求设计的一个 Java Bean，若具有图形化的描述能力，则可以直接在 GUI 设计中作为一个可视的组件进行使用。另外，Java 允许 Java Bean 以一种十分方便的方式对类的对象进行存储和载入。这里以 Point 类为例进行简单的说明，其定义如代码 9.2 所示。

代码 9.2　一个简单的 Java Bean

```
public class Point implements Serializable{
    private int x;
    private int y;
    public Point(){
        this.x=0;
        this.y=0;
    }
    public Point(int x, int y){
```

```
        this.x=x;
        this.y=y;
    }
    public int getX(){
        return x;
    }
    public void setX(int x){
        this.x=x;
    }
    public int getY(){
        return y;
    }
    public void setY(int y){
        this.y=y;
    }
    @Override
    public String toString(){
        return "[" + x + "," + y + "]";
    }
}
```


代码 9.3 说明如何使用这个 Point 类。

代码 9.3　对 Java Bean 的使用

```
import java.beans.XMLDecoder;
import java.beans.XMLEncoder;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;

public class BeanClient{
    public static void main(String[] s){
    Point p1= new Point();
    Point p2= new Point(3,4);
    Point p3= new Point(1,1);
    Point p4= new Point(2,2);
    String data="point.txt";
    try{
        XMLEncoder out= new XMLEncoder(new BufferedOutputStream(new FileOutputStream
                    (data)));
        out.writeObject(p1); out.writeObject(p2);
        out.close();
    } catch (FileNotFoundException e){}
    try{
        XMLDecoder in= new XMLDecoder(new BufferedInputStream(new FileInputStream
```

```
                        (data)));
            p3= ((Point)in.readObject());
            p4= ((Point)in.readObject());
            in.close();
        } catch (FileNotFoundException e){}
            System.out.println(p3 + " " + p4);
        }
    }
```

在上面的程序中，首先创建了 4 个 Point 对象，然后使用了类库中用于对文件进行处理的 XMLEncoder 类及其他辅助类，并将 p1、p2 两个 Point 对象以 XML 格式写入文件 point.txt 中。在第二个 try-catch 程序块中又将刚才存储到文件中的两个 Point 对象以 XML 格式再次读出并格式化输出。程序提供了一种对象的简单存储方式，同时该存储也具有很好的阅读性，并方便第三方的 XML 工具进一步处理。

需要说明的是，对于 XMLDecoder 对象，当进行数据读取时，能够自动使用与 XML 文件元素中的属性名字对应的 set 方法对实例变量进行赋值，这个过程的控制流程实际上是反转的，因为是对组件方法的间接调用。Java 中使用一种反射的技术，可以在运行时决定一个对象的哪些方法被使用。这里其实就使用了这种反射机制，首先在 writeObject 中寻找与传递对象中属性名字相符合的 get 和 set 方法，然后利用它们以 XML 的形式编码对应的属性。这时如果将 Point 类中的 getX() 方法注释掉，程序将失去对属性 x 处理的能力，而只能自动存储 y 的值。

9.4.2 框架

框架，也称容器，也是一种程序轮子。在程序实现上，人们一直在追求一种更为简便和系统化的实现方法。在这种方法中，并不需要亲自从零来开发构成程序的一些组件，也不必亦步亦趋地完成开发的每个环节，而是能够更快速、高效和正确地将很多原始的工作积累合成到一个更大粒度的、半成品式的系统中，开发者只需对它进行必要的参数定制，就能够将其打造成符合用户需求的真实系统，这就是框架的作用。

类库和函数库试图提供一套完整的功能，但这些功能通常都是分散的，需要开发者根据需要进行选择和拼凑。一个组件也试图提供一个完整的功能，不过，此功能通常只限定在一组类中，因此具有较小的范围和粒度。除了上述两种可以进行重用的技术，人们也在寻找基于两者基础之上的，由逻辑上相关的若干类在一起完成某种统一功能的子系统，如负责对象在数据库中的持久化，从数据库中读取并重建对象（Object Relation Mapping，ORM）的一个相对独立的功能体。

框架定义了对象的规则要求，以能够对它们进行统一的组织和管理，这通常是通过接口和继承的方式实现的。当框架中嵌入了满足要求的业务对象后，框架的功能则会融入具体的业务类，使得提供的功能得以展现。框架实际上是某种应用的半成品，是一组供选择的组件，用以嵌入并定制具体的系统。简单来说，框架是别人搭好的舞台，由使用者来做表演。组件与框架最主要的差别就是控制权在框架中要进行转移。也就是说，框架中的类会调用那些由用户补充实现的对象中的方法，而不会反过来。但在组件中，是会通过反射的方式发生的。

Enterprise Java Beans（EJB）和 Spring 是两个功能强大的框架，其主要功能是使对象的管理及与

数据库的交互更加容易。框架的思想在 Java 的类库中也有一定程度的应用，考虑一个实际的例子，使用 Java 中 JTree 视图来描述一个树状结构的项目信息，显示的效果如图 9.6 所示。

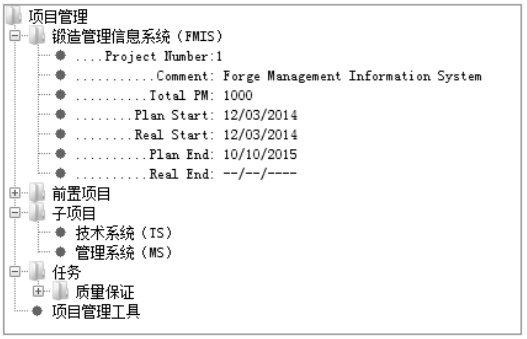


图 9.6　树状显示项目信息

图 9.6 中，项目、子项目及前置项目等信息以层次结构进行显示。Java 语言中提供了使用树结构进行描述的若干个类和接口，它们可以对树结构进行直接显示或操作控制，如树的某个节点的展开或合并等。为了使用树的框架，在 Java 中需要一个类对 TreeModel 接口进行实现，此接口的主要内容如表 9.1 所示。

表 9.1　Java 类 TreeModel 中的主要方法

方　法	返回类型	描　述
addTreeModelListener（TreeModelListener）	void	为树改变后的事件 TreeModelEvent 添加监听器
getChild（Object parent, int index）	Object	返回对象 parent 的孩子中索引为 index 的孩子
getChildCount（Object parent）	int	返回对象 parent 的孩子数
getIndexOfChild（Object parent, Object child）	int	返回对象 parent 中孩子为 child 的索引值
getRoot（）	Object	返回树的根对象
isLeaf（Object node）	boolean	返回对象 node 是否为树的叶子节点
removeTreeModelListener（TreeModelListener）	void	删除某个之前使用方法 addTreeModelListener 加入的监听器
valueForPathChanged（TreePath path, Object newValue）	void	用户将 path 标识的项的值更改为 newValue 时，进行通知

图 9.7 中通过类图描述了该接口的具体使用方法。首先，图中存在一个 ProjectTreeModel 类，对 TreeModel 接口进行了实现，ProjectTreeModel 类与项目管理类 ProjectMan 具有关联关系。注意到两个类之间的关联是双向导航的，即 ProjectMan 类也能够识别 ProjectTreeModel 类，这里实际使用的是观察者模式。另外，还提供了一个 NodeInfo 接口，是节点信息的抽象描述，ProjectMan 类和 Project 类分别对其进行了实现。主要的实现如代码 9.4 所示。

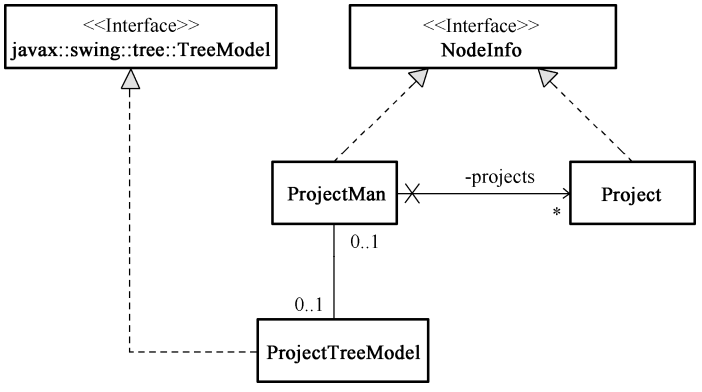


图 9.7　Java Tree 框架类图

代码 9.4　Java Tree 框架的实现

```
//ProjectTreeModel.java(部分)
public class ProjectTreeModel implements TreeModel{
    private ProjectMan pm;
    private List<TreeModelListener> listener=
                            new ArrayList<TreeModelListener>();
    public ProjectTreeModel(){
        pm = new ProjectMan();
    }
    public Object getRoot(){
        return pm;
    }
    public Object getChild(Object arg, int i){
        return ((NodeInfo)arg).ithElement(i);
    }
    public int getChildCount(Object arg){
        return ((NodeInfo)arg).numElements();
    }
    public void addTreeModelListener(TreeModelListener arg){
        listener.add(arg);
    }
}
//NodeInfo.java
public interface NodeInfo{
    public int numElements();              //返回所有元素个数
    public NodeInfo ithElement(int i);     //返回第 i 个元素
    public boolean isEmpty();
    public int atPosition(NodeInfo ni);
    public String title();
}
//ProjectMan.java(部分)
public class ProjectMan implements NodeInfo{
    private List<Project> projects = new ArrayList<Project>();
    public int numElements(){
        return projects.size();
    }
    public NodeInfo ithElement(int i){
        return projects.get(i);
    }
    public boolean isEmpty(){
        return false;
    }
    public String title(){
        return "项目管理";
```

```
    }
    @Override
    public String toString(){
        return title();
    }
}
//Project.java(部分)
public class Project implements NodeInfo{
    private Attribute[] attr = new Attribute[8];
    private TreeList<Project> subprojects = new TreeList<>("子项目");
    private TreeList<Project> preprojects = new TreeList<>("前置项目");
    private TreeList<ProjectTask> tasks = new TreeList<>("任务");
    public int numElements(){
        return attr.length+3;
    }
    public NodeInfo ithElement(int i){
        switch(i){
            case 0: case 1: case 2: case 3: case 4: case 5: case 6:
            case 7: return attr[i];
            case 8: return preprojects;
            case 9: return subprojects;
            case 10: return tasks;
        }
        return null;
    }
}
```


ProjectTreeModel 类的主要任务是对 TreeModel 接口中方法进行实现。因为 ProjectMan 类中存有项目真正的树状层次信息，所有 TreeModel 接口中的相关方法都是通过访问 ProjectMan 类的对象获取。对于 Project 类来说，一方面受 ProjectMan 对象的控制和管理，另一方面又是项目树结构中的内容，所以，对于 ProjectTreeModel 类的几个调用请求，会被 ProjectMan 类转发给 Project 类。为了使 ProjectTreeModel 类的职责尽量简单化，树描述中的所有类都需要实现一个统一的接口，即 NodeInfo。这个接口保证了 ProjectTreeModel 类对于树中所有的节点，以统一的形式进行处理。

另外一种可选的方法是相关的类不去实现 NodeInfo 接口，而直接实现 TreeModel 接口，这可以使得设计更为灵活。但是由于该接口的方法，这些类及属性并不都会需要，因此这里还是补充了一个单独的 NodeInfo 接口。为了能够将 ProjectTreeModel 类的信息展现出来，需要配合使用一个与显示相关的视图对象 JTree，其相关的代码包含在代码 9.5 中。

代码 9.5　Java Tree 框架的显示

```
ProjectTreeModel ptm = new ProjectTreeModel();
JTree tree = new JTree(ptm);
JScrollPane scroller = new JScrollPane(tree);
add(scroller, BorderLayout.CENTER);
```

9.5 数据的持久化

数据是软件不可分割的一部分，当业务系统停止运行后，需要将相关的业务数据存储起来，以对未来的使用需要提供支持。相对于应用运行时存在于内存中的“瞬时”数据，将那些永久保存的数据称为“持久化”的数据。

数据的持久化存储总的来说有两种方式，实际可根据项目的特点和要求进行选择。一种方式是应用将数据直接存储于物理文件中，另一种方式则是通过专门的存储系统对业务数据进行保存，如数据库系统。使用数据库系统最大的好处在于不同的应用可以在同一时间对同一数据并发使用，提供了很好的数据共享性和安全性。

9.5.1 文件持久化

对于直接使用文件的方式进行数据的持久化，不同的程序语言提供了不同的技术支持。其中一个基本功能是能够让开发者将数据以二进制的原始数据形式存储于文件中。基于这个基本的存储功能，可以衍生出更多的、方便的文件存储服务。这里主要以 Java 语言为例，对使用文件存储的方式进行简单讨论。代码 9.6 描述了将数据以二进制流的形式直接在文件中写入的方法。

代码 9.6 简单文件存储示例

```
FileOutputStream o = new FileOutputStream("c:\\data.txt");
String s = "Hello again";
for (int i=0; i<s.length(); i++)
   o.write(s.charAt(i));
o.close();
```

上述的方法对写入的字符有一定的限制，效率也不高。改进的方法可以考虑首先将数据在缓存中收集，然后一次性地写入磁盘。在 Java 中，可以使用缓存类来达到这个目的，如代码 9.7 所示。

代码 9.7 带缓存的文件存储示例

```
BufferedOutputStream o = new BufferedOutputStream(
                         new FileOutputStream("c:\\data.txt"),4);
String s="Hello again";
for (int i=0; i<s.length(); i++)
   o.write(s.charAt(i));
o.close();
```

更进一步，类似 DataOutputStream 的类还提供一些对标准数据类型的完整值的写入操作，如方法 writeInt、writeString 等。这对于对象意味着，如果要进行读写操作，需要对所含有的每个实例变量单独进行，如代码 9.8 所示。

代码 9.8　使用数据类型进行文件存储

```
public class Point{
    private int x;
    private int y;
    public Point( int a, int b){x=a; y=b;}
    public void save(DataOutputStream o)throws IOException{
        o.writeInt(x);
        o.writeInt(y);
    }
    public static Point read(DataInputStream i)throws IOException{
        return new Point(i.readInt(), i.readInt());
    }
}
```

创建文件对数据的存储方式的好处是可读性比较好，可以方便地通过手工编辑或者其他程序进行读取。这种方法显著的问题是开发者必须为每个需要持久化的类编写存储或读入的方法，这是一项很烦琐的工作。所以，一种更直接的想法是将类作为一个整体进行存储。在 Java 中，可以对进行持久化的类实现 Serializable 接口，尽管此接口不含有任何方法，只需实现即可。持久化类要求含有的所有实例变量的类型也都实现了 Serializable 接口，否则该实例变量不能被持久化。如果不需要对某个实例变量持久化，需要在其前边标注 transient 说明，这在网络应用中非常有用，它可以有效降低传输的数据量。代码 9.9 给出了该方法的一个示例。

代码 9.9　通过 Serializable 接口进行文件存储

```
public class Person implements Serializable{
    private String lname;
    private String fname;
    transient private String address;
}
//...
Person p = new Person ("Ohst", "Dirk");
ObjectOutputStream out = new ObjectOutputStream (
                          new FileOutputStream("test"));
out.writeObject(p);
//...
ObjectlnputStream in = new ObjectInputStream (
                        new FileInputStream ("test"));
p = (Person)in.readObject();
```

这个方法的缺点是，存储的文件是二进制形式的，只能由该类的应用识别和读取。而且，当程序中的类发生某些改变时，如增加了一个新的实例变量，再次读取时就会出现问题。除了二进制文件的存储，Java 语言还能使用 XML 文件的形式进行数据的存储。大数据量的存储一般情况下都要借助数据库系统来进行。

9.5.2　数据库持久化

经常使用的与数据库交互的方式是直接将 SQL 指令作为字符串从 Java 发送到数据库中执

行。JDBC(Java DataBase Connectivity，Java 数据库连接)是一种用于执行 SQL 语句的 Java API，代码 9.10 说明了 JDBC 的工作方式。

代码 9.10　通过 JDBC 与数据库交互

```
//Load Oracle JDBC driver
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
//Connect to DB
Connection conn =
    DriverManager.getConnection(
                  "jdbc:oracle:thin:@192.168.1.120:1521:DBName",
                  "user", "password");
//Create a class for DB operation
Statement stmt = conn.createStatement();
//Query DB
ResultSet rs = stmt.executeQuery("SELECT * FROM EMP");
//Output
while(rs.next()){
    int id = rs.GetInt("EMPNo");
    String name = (String)(rs.getObject());
    System.out.println("Num " + id + ": " + name);
}
```


上面的代码分为两个部分，第一部分建立与数据库的连接，这是通过抽象工厂模式实现的，DriverManager 类是抽象工厂类。第二部分是运行一个 SQL 指令，然后使用 Java 将查询后的结果显示出来。这里主要的类是 ResultSet 类，通过它逐步将数据库系统返回的结果读出。这个类的工作方式类似于迭代器 Iterator，每次迭代都检查是否存在后续的元素并步进当前记录的指针，然后逐条读出数据，JDBC 的具体使用方法依赖于每个数据库厂商提供的 JDBC 驱动版本，各数据库中可能会有差别。另外，JDBC 也支持通过方法 conn.commit()和 conn.rollback()实现对数据库中事务的控制。

新系统的开发，开发者大多会选择关系型数据库实现对数据的管理，因为关系型数据库经过了多年的实践考验并具有较大的市场份额，能够适应不同的开发需求。关系型数据库较为明显的缺点是，业务存储的模型需要事先在数据库端设计和构建，在数据持久层中需要在应用中进行一些程序设计工作及对象数据与关系型的兼容性处理等。这种不兼容性又称为阻抗失配(Impendence Mismatch)，它的解决方式一般是使用单独的中间件(框架)对应用中存在的对象与数据库存储过程进行自动化管理，即所谓的对象关系映射(OR Mapping)，其目的是让开发者尽可能少地关注数据库的关系模型及其实现细节。类似的框架包括 EJB、Hibernate、Mybatis 及 JDO 等，它们的共同点是能够高效地管理实体类与其持久化形式间的转换，包括对它们参与的事务的管理，使开发者可以专心地致力于业务功能的开发，而不必处理对象关系转换这些烦琐事情。

9.6　领域特定语言

编程语言的发展趋势是较高的抽象程度及多种结构构成方式。高级语言具有的优势是抽

象级别的提高及对软件工程概念的支持，如高级语言扩展了现有的指令集，加入模块和类等新的程序构造结构，使更为复杂的程序开发成为可能。

如果人们只是希望使用某种编程语言来实现某些特定的相似任务，那么可以考虑构建更复杂的命令。例如，对于菜单结构的描述，第一级是主菜单，其内具有二级菜单项并对它们进行类似的嵌套组织。尽管一些典型的编程语言大多通过类库和框架等机制对菜单的实现进行支持，但要应用到实际项目中还是要做一些编码工作。为了减少菜单的开发时间和开发成本，可以考虑使用某种抽象语言来提升设计，使开发者不依赖编程语言而快速进行菜单的结构描述。

这就是领域特定语言(Domain Specific Language, DSL)的初衷，为不同的领域补充特定的、不依赖具体编程语言的抽象指令，或者说，针对某一特定领域具有受限表达的一种程序设计语言。例如，对上面的菜单结构，可以使用如下的表达方式：

```
<menu_level> ::= Menu( <name_1> <submenu>, ..., <name_n> <submenu> )
<submenu> ::= <menu_level> | <action_id>
```

这是对菜单这个特定的领域设计的一种特殊的表达结构。受限性是相对于通用的编程语言而言的，即以上的 DSL 只是为“菜单”这个领域服务，不具有通用性。下面的语句则可以用来构建一个具体的菜单结构：

```
M = Menu("Data" Menu("Load" 1, "Save" 2, "Save as..." 3), "Format" Menu("Font"
Menu("Larger" 4, "Smaller" 5), "Alignment" Menu("Center" 6, "Left" 7), "Standard"
8), "Help" 9);
```

对应的菜单结果如图 9.8 所示。<action_id>是指对每个菜单项的单击所触发的动作标识，可以对应到具体方法的调用。实践中，一般使用 DSL 作为概念上的表示，而在实现上常采用 XML。

DSL 的核心价值在于，它提供了一种手段，可以更加清晰地就系统某部分的意图进行表达和沟通。软件项目中最困难的部分，就是开发者与软件用户之间的沟通，DSL 提供了一种清晰而准确的概念性的描述语言，可以有效地改善这种沟通。

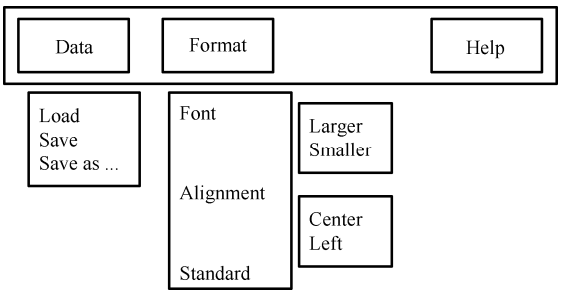


图 9.8　使用 DSL 建立的示例菜单

DSL 的另一个比较常用的业务领域是对 Web Service 的描述，因为除了对实际服务的开发，还存在很多需要配置的任务，如使用 UDDI 和 WSDL 等与编程语言无关的服务描述语言对编译后的程序在何处存放及这些程序如何通过登记服务进行注册等的说明。在这种情况下，我们希望将代码运行于不同的环境中，这类理由也是使用 DSL 常见的原因之一，即满足平台无关性的需求。

DSL 的应用可以通过两个不同的工作过程进行，分别为 DSL 的建立过程和 DSL 的使用过程，如图 9.9 所示。首先，具有领域经验的领域开发者对业务进行抽象并通过 DSL 概念化，尽可能将相似的业务步骤抽象为简单的 DSL 指令，在这个过程中，重要的是对每个命令的语义能够进行精确的描述。当然，重新定义一个全新的语言体系是比较困难的(外部 DSL)，因此另外的方式是借助某种熟悉的语言对业务命令进行组织和定义(内部 DSL)。

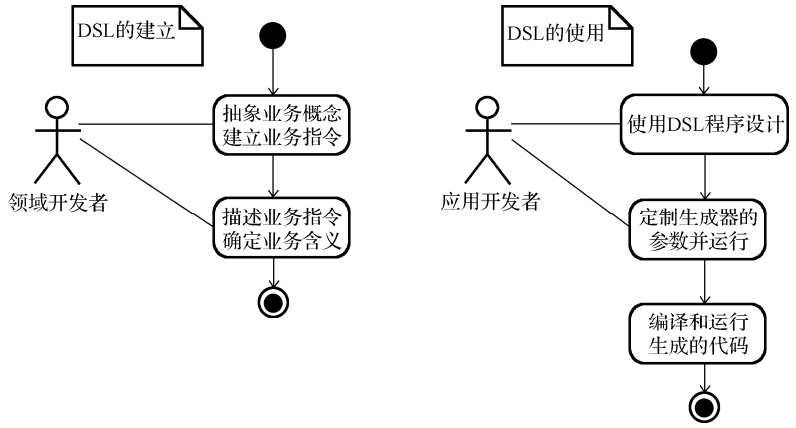


图 9.9　使用 DSL 的工作过程

外部 DSL 是一种“不同于应用系统主要使用语言”的语言。外部 DSL 通常采用自定义语法。不过，选择其他语言的语法有时候也很常见(如 XML)。宿主应用的代码会采用文本解析技术对使用外部 DSL 编写的脚本进行解析，一些传统 UNIX 中的 Shell 语言就符合这种风格。可能经常会遇到的外部 DSL 包括正则表达式、SQL、AWK 及像 Spring 和 Hibernate 这样的系统使用的 XML 配置文件。

内部 DSL 是一种通用语言的特定用法。用内部 DSL 写成的脚本是一段合法的程序，但是它具有特定的风格，而且只用到语言的一部分特性，通常用于处理整个系统一个小方面的问题。用这种 DSL 写出的程序有一种自定义语言的风格，与其所使用的宿主语言有所区别。这方面最经典的例子是 Lisp。Lisp 开发者编写的程序就是创建和使用 DSL。另外，前面介绍过的程序轮子框架也可以视为一种内部 DSL，如 Spring MVC 框架就是基于 Java 的用于 Web 开发领域的内部 DSL。

总之，DSL 提供了一种抽象，我们可以基于这种抽象来思考领域问题。这种抽象非常有价值，可以更容易地表述领域行为，效果远胜于依据底层构造进行的思考。但 DSL 同样也存在一些问题，如学习成本、DSL 本身的构建及维护成本等。

9.7　模型驱动架构

程序轮子实际上就是可重用思想的体现，这里所指的可重用性主要针对的是对程序代码的重用。模型驱动架构(MDA)的基本思想是提供一种正式的解决方案，其与具体编程语言，甚至是架构无关。

9.7.1　原理及过程

使用 MDA 进行开发的过程可以分为 4 个阶段。

(1) CIM (Computation Independent Model)：聚焦于系统环境及需求，但不涉及系统内部的结构与运作细节。

(2) PIM (Platform Independent Model)：聚焦于系统内部细节，但不涉及实现系统的具体平台。

(3) PSM (Platform Specific Model)：聚焦于系统落实于特定具体平台的细节，如 EJB、J2EE 或.NET (都是一种具体平台)。

(4) Coding：最后依据 PSM 的 UML 模型内容，按图施工，编写出适用于特定具体平台的代码。

MDA 描述的软件开发生命周期和传统生命周期没有大的不同，主要的区别在于开发过程创建的软件制品，包括 PIM、PSM 和代码。PIM 是具有高抽象层次、独立于任何实现技术的模型，PIM 被转换为一个或多个 PSM。PSM 是为某种特定实现技术量身而做的模型，如在 EJB 中，PSM 是用 EJB 结构表达的系统模型。开发的最后是把每个 PSM 转换为代码，PSM 同应用技术密切相关。

传统的开发过程从模型到模型的变换，或者从模型到代码的变换是手工完成的。但是，MDA 的变换都是由工具自动完成的。从 PIM 到 PSM，再从 PSM 到代码，都可以由工具实现。这里 PIM、PSM 和 Coding 模型作为软件开发生命周期中的设计制品，而在传统的开发方式中它们是文档和图表。重要的是，它们代表了对系统不同层次的抽象，从不同的视角来看系统，将高层次的 PIM 转换到 PSM 的能力提升了抽象的层次，使开发者更加清晰地了解系统的整个架构，而不会被具体的实现技术所“污染”。同时，对于复杂系统，也减少了开发者的工作量。

在这个转换的过程中，模型间转换规则的定义也非常重要，即要对模型间的过渡进行形式化的描述。例如，在 PIM 中的某个集合类型 Collection，可以对应 Java 中的 ArrayList 或者 C++的 std::list 类型。MDA 希望能制订出各式独特的具体平台专属的 PSM 转换规则，并且最好可以由厂商配合设计出 MDA 开发工具，以便能够使中立的 PIM 自动转出特定平台的 PSM 模型。

图 9.10 对整个的转换过程做了一个总结，将 PIM 和 PSM 的转换描述为模型 1 到模型 2 的过渡。所有模型和转换规则需要正式的语义定义，因此，OMG (Object Management Group，对象管理组织) 出台了一系列的规范将 MDA 方法标准化，其中最重要的语义描述包含在规范 Meta Object Facility (MOF) 中。

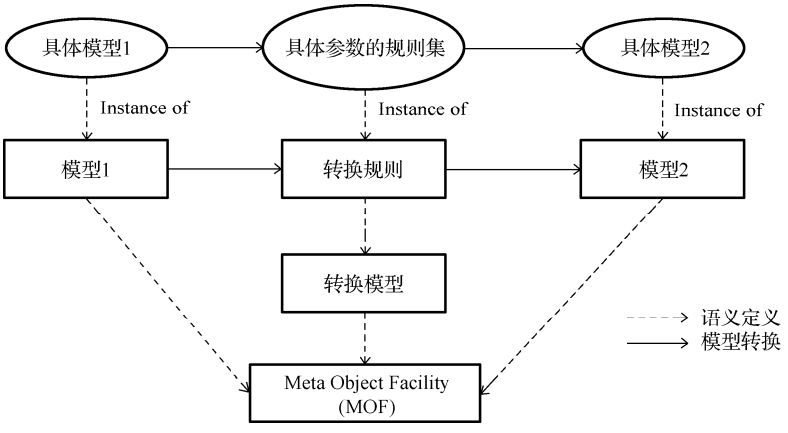


图 9.10　MDA 转换过程

基于 MDA 的开发流程如图 9.11 所示。对具体问题的解决，一开始要尽可能地通过中立的方式进行构建，然后在接下来的步骤中逐步对抽象的元素进行具体化，这是一个迭代的过程，直到最终得到能够运行的程序。

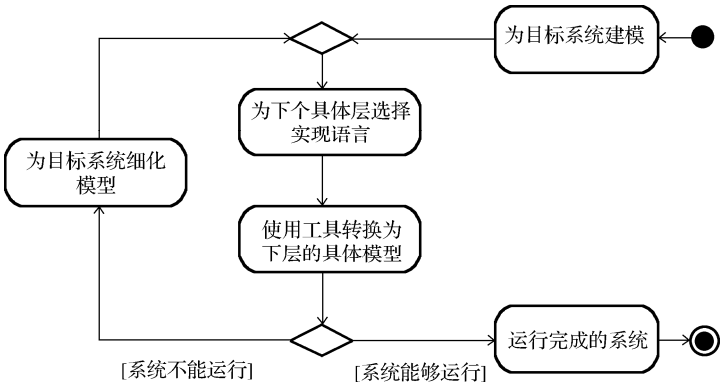


图 9.11　MDA 开发流程

9.7.2　MDA 的应用

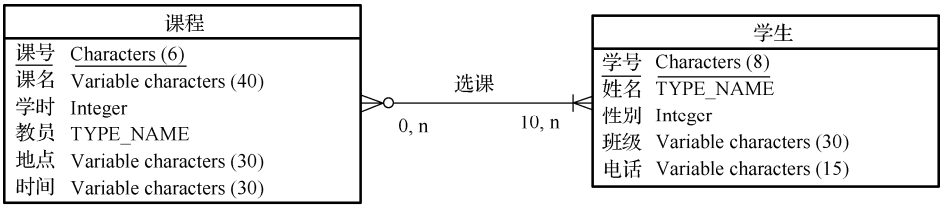
MDA 的一个具体应用是在数据库设计工具 PowerDesigner 中针对实体关系图(ERD)的构建。数据库模型的设计依次分为 4 个主要的阶段，分别为概念设计模型(Conceptual Design Model，CDM)、逻辑设计模型(Logical Design Model，LDM)、物理设计模型(Physical Design Model，PDM)和数据库代码，分别对应为 MDA 中的 CIM、PIM、PSM 和 Coding 4 个层次。例如，学生选课系统中的 CDM、LDM 及 Oracle 的 PDM 对应的 3 个模型如图 9.12 所示，其中模型 CDM、LDM 是与具体数据库平台无关的模型，含有的字段类型也是中立类型，并由设计工具负责维护到具体数据库平台的映射规则。PDM 是在 Oracle 数据库平台上对应的物理模型，包含有具体的实现信息，如平台相关的列类型及约束信息等。

最后对应生成的 Oracle 模型代码如代码 9.11 所示。

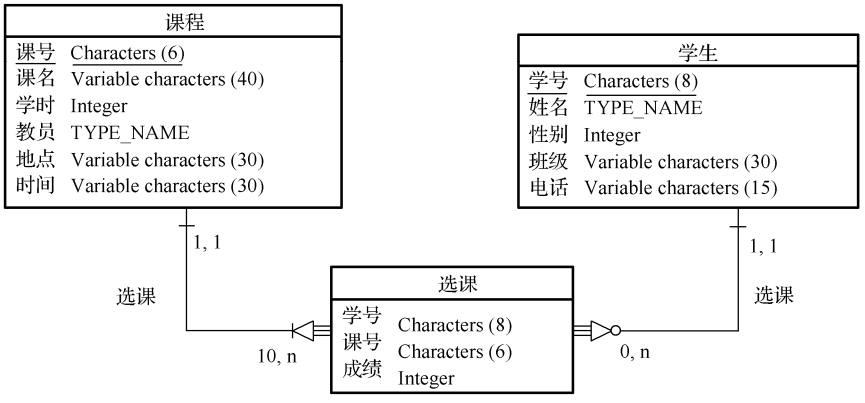
代码 9.11　Oracle 物理模型的实现

```
create table Course (
    CNo CHAR(6)not null,
    CName VARCHAR2(40)not null,
    Hours INTEGER not null,
    Lecturer VARCHAR2(10)not null,
    Room VARCHAR2(30)not null,
    Time VARCHAR2(30)not null,
    constraint PK_COURSE primary key (CNo)
);
create table Student (
    SNo CHAR(8)not null,
    SName VARCHAR2(10)not null,
    Gender INTEGER default 0 not null
    constraint CKC_GENDER_STUDENT check (Gender in (0,1)),
    Class VARCHAR2(30)not null,
    TelNumber VARCHAR2(15),
    constraint PK_STUDENT primary key (SNo)
);
create table slcourse (
    SNo CHAR(8)not null,
```

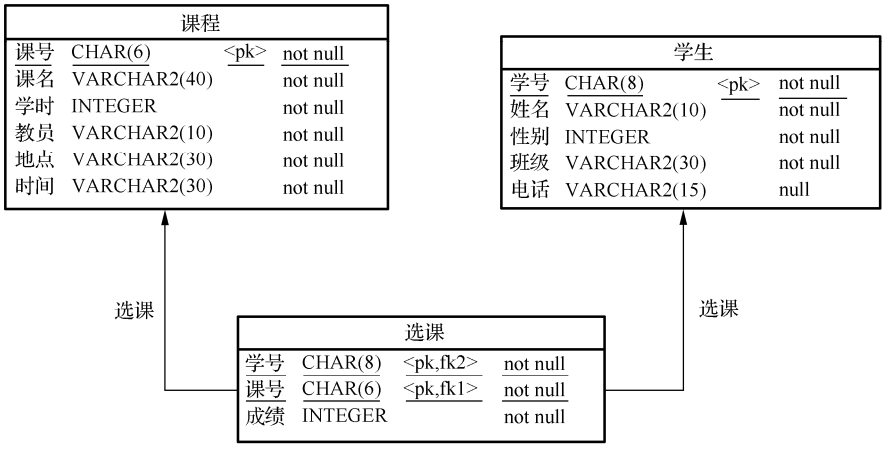
```
    CNo CHAR(6)not null,
    Score INTEGER not null,
    constraint PK_SLCOURSE primary key (SNo, CNo)
);
/* 此处省略索引的创建 */
/* 以下为外键约束 */
alter table slcourse add constraint FK_SLCOURSE_STUDENT foreign key (SNo)
    references Student (SNo);
alter table slcourse add constraint FK_SLCOURSE_COURSE foreign key (CNo)
    references Course (CNo);
```



(a)概念设计模型CDM



(b) 逻辑设计模型LDM



(c) Oracle物理设计模型PDM

图 9.12　数据库模型设计的 MDA 应用

9.8 重构

一个系统的开发是逐步完成的，不可能一蹴而就。在开发的过程中，需要不断地进行决策上的选择及功能上的演化，因此，一个客观趋势就是程序会变得越来越复杂。程序的可重构性(Refactoring)是面向对象的目标之一，为了保证这个目标的实现，必须要尽可能地对程序进行简化，以增加其可理解性和可维护性。重构前的第一件事是要准备一系列的测试，因为重构首先要确保目标的行为不发生改变，使用测试用例可以自动化地对这一要求不断地进行确认，同时又帮助发现缺陷。

重构的作用首先是帮助开发者改善软件的设计。程序的设计一般会随着时间而退化，因为设计的结果常常只顾当前的需求，而需求的变化引来了程序代码的变化，因此代码结构由此趋于不可控。例如，退化的表现之一就是分散在代码各处的重复代码，通过重构使得代码对同一件事情说一遍并且只说一遍，这也是一个好的设计的精华所在。

重构还能够帮助理解代码。程序维护人员可能会花上一周的时间来理解程序，修正一处问题也许只需要 10 分钟，其原因就是程序开发者在开始的时候就没有考虑为未来的维护做准备。甚至时过境迁，开发者自己也会忘记自己编写的代码的含义。重构通过对程序结构的优化帮助理解程序。

下面通过一些简单的例子来说明重构的具体活动。例如，编码过程需要符合约定的风格，可能的编码规则如下：

(1)方法的名字要尽可能地“见名知义”;

(2)方法最长应不超过 12 行，尽可能少地包含 while、switch 和 if 逻辑块。

第(1)条要求为方法指定一个合适的名字，这能够较显著地增加代码的可读性。例如，代码 9.12 中的方法名就比较合适，因为名字本身就对方法的作用进行了解释。当然，该方法内部实现的结构还可以继续优化，局部变量 smoker 的引入使其与下层模块的耦合性大大增加，这里可考虑使用运行时的多态思想对程序结构进一步调整。

代码 9.12　容易理解的方法命名

```
public int riskCompute(int age, boolean smoker){
    int result = basisCompute(age);
    if (smoker)
        result += smokerExtra(age);
    return result;
}
```

由此引出了另外一个不容忽视的问题，就是如何对复杂的方法进行简单化。为此，代码重构发挥了作用，它给出了一些系统化的针对方法级别的简化方法。借助重构可以在不改变代码行为的基础上，对代码的内部结构进行优化，对提升代码的质量具有重要意义。

重构可以发生在不同的级别上，如在类方法的级别上，主要考虑的是通过拆分或合并等做法将语法正确的代码片段整合为新的方法；在类的级别上，考虑更多的是类的独立性和耦合性等。

代码 9.13 所示的重构前的代码块中被阴影标记的部分应该提取出来放入一个新的方法中，通过对新方法的调用返回一个结果，并使用一个本地变量记录。这样做的主要目的是将这段具有控制耦合的代码消除掉。

代码 9.13　重构示例代码 1

```
//重构前
public int ref ( int x, int y, int z){
    int a=0;
    if (x>0){
        a=x;
        x++;
        --y;
        a = a + y + z;
    }
    return a;
}
//重构后
public int ref ( int x, int y, int z){
    int a=0;
    if(x>0){
        a = doit(x, y, z);
    }
    return a;
}
private int doit (int x, int y, int z){
    int a;
    a=x;
    x++;
    --y;
    a=a+y+z;
    return a;
}
```


代码 9.14 中重构前的代码，使用 Java 进行重构要稍微复杂一些，因为被阴影标记部分的代码块对 3 个本地变量同时进行修改，这在分离的方法中不能直接实现，会增加实现上的复杂度，但 C++可以很好地处理，如代码 9.14 中重构后的代码所示。

代码 9.14　重构示例代码 2

```
//重构前
public int ref2 (int x){
    int a=0;
    int b=0;
    int c=0;
    if(x>0){
        a = x;
        b = x;
        c = x;
    }
    return a+b+c;
```

```
}
//使用 C++重构后
int Computation::ref2 (int x){
    int a=0, b=0, c=0;
    if(x>0){
        abcModify(a,b,c,x);
    }
    return a+b+c;
}
void Computation::abcModify(int& a, int& b, int& c, int x){
    a=x; b=x; c=x;
}
```

重构的时机可以选择在代码发生变化的时候，如添加新功能的时候、修正缺陷的时候或者代码被评审的时候。

9.9 习题

1．以下 XML 文档是 well-formed 的吗？请指出错误并加以改正。

```
<?xml version="1.0" encoding="GB2312"?>
<user id=1>
    <Name>John</name>
    <password>123
    <roles><role>admin</roles></role>
</user>
<user id=2>
    <name>Mike</name>
    <password>abc
    <roles has="guest" has="buyer"></roles>
</user>
```

2．对于以下给定的 XML 文档，编写一段 Java 程序，分别采用 DOM 和 SAX 两种方式计算所有书目的价格之和。

```
<?xml version="1.0" encoding="utf-8"?>
<booklist>
    <item>
        <code>16-048</code>
        <category>scripting</category>
        <release_date>2018-03-25</release_date>
        <title>Java Insight</title>
        <price currency="usd">43.56</price>
    </item>
    <!-- 省略余下部分 -->
</booklist>
```

3．针对上题中使用 XML 格式的数据，设计关系模型并保存在数据库中，使用 Java 通过 JDBC 计算上题中要求的结果。

第10章　交 互 设 计

本质上，软件的最终目的是直接为人类服务，即软件应具有可用性。软件的可用性能决定整个项目的成败，因为即使软件功能正确，但操作性和用户体验都很差，通常用户也是很难接受的。软件的可用性和可操作性对于开发者来说也许过于抽象和难以控制，本章的目的就是针对交互设计和可用性进行介绍性的说明。

本章首先对交互设计、可用性及其背景进行概述，然后逐步细化地讨论可用性需求的构成要素和实现方式，最后给出对可用性进行验证的方法。

10.1　交互设计概述

起初，交互设计对软件可用性的影响并未得到开发者足够的重视，计算机科学的核心领域也未涵盖可用性的内容，因为可用性涉及的范围较广，涵盖多个不同学科的内容。可用性主要涉及的领域如下。

(1) 设计心理学：设计心理学中有一项任务是研究颜色和形状带来的影响。颜色和形状的选择总是要结合具体环境进行考虑。例如，暗色调通常可以表示严谨或严肃的气氛，网页中的暗色背景也可以传达一种神秘或者哀思。另外，需要注意的是，颜色在不同的文化中也可能有着完全不同的含义。例如，白色在西方国家的文化中多代表纯洁之意，但在亚洲大多数国家中则多表示哀悼之情。除了单一的颜色，还需要留意一些颜色的组合情况，如白色背景上蓝色的文字要比绿色背景上黄色的文字读起来舒服得多。

除颜色外，形状及其布局在界面设计中也有着重要的影响。相同的颜色及边框能够将逻辑上相近的功能拉近或进行视觉上的分组，与其他功能形成对比并区分开来。另外，形状也可以为观察者带来情绪上的影响，如粗线条会使得人具有粗糙的感觉，而细线条则显得高贵和细致。总之，设计心理学最初为广告设计领域提供了指导和规范，其中大部分在软件的界面设计中也同样适用。

(2) 人体工程学：人体工程学的一部分工作是对工作空间进行设计，其中一项最主要的需求就是工作环境和工作设备要适合工作。软件的使用者来自不同的领域，他们的职业和角色也不尽相同，软件对于他们来说是工作中不可或缺的工具，但由于软件使用者往往并不直接参与软件系统的开发，因此必须要提供给他们一个直观、方便、可以与计算机系统协同工作的界面。

在软件系统开发的过程中，首先通过对业务工作流的分析，完成软件的需求分析，这是一个非常重要的环节，通过需求分析，能够明确开发的具体任务。接下来，要围绕这些业务流程，设计出合理的图形界面，用以有效组织起这些实实在在的具体功能，这也是交互设计的主要内容。另外，要注意软件中的主要功能应易于访问，误操作或错误能够得到很好的捕获，并给出有针对性的提示，使用户不会淹没在信息的海洋中。

(3)软件人体工程学：作为计算机科学的一个学科，软件人体工程学主要是以上提到的想法在实际软件中的实现，可以认为是在已有的设计思想上加入了特殊的边界条件。若通过某人机交互类库来实现交互的界面，则可能不是所有理论上可行的交互技术都会得到支持和使用。软件人体工程学的目的就是在进行人体工程的实现过程中，在技术上和经济上提供保证。

另外，进行界面设计时可能还需要考虑一些对软件人体工程有影响的细节，如客户的公司可能要求在界面中以某种方式嵌入其商标(Logo)或者使用其公司要求的颜色等，这是市场的要求，虽然在技术上可能认为红色背景下的白色文字让人阅读起来不是最佳搭配，但对很多使用者来说，会将这样的组合与具体公司联系起来。

另一个不容小觑的界面设计因素是时尚。从 20 世纪 90 年代开始，视窗系统盛行，窗口提供了一种图形化的信息展现形式，其右上角设置了一些直接受操作系统控制的按钮，如窗口的最大(小)化、关闭等。在标题栏一般设置能够展开的菜单，每级菜单可以继续包含子菜单或子菜单项，通过单击它们执行具体的功能。多年来，视窗系统得到了广泛的应用，但也受到了一些人体工程学方面的批评，而且窗口的千篇一律会给很多用户带来审美疲劳，影响用户体验。因此，在窗口的设计上又发展出一些多样性的处理方式，如圆角的窗口和半透明的窗口，由此也出现了一些专门的窗口设计工具，供有经验的开发者使用。还衍生出了另外一些基本的界面设计理念，如允许用户动态地根据喜好来改变界面元素的布局或形状等，又称“皮肤”。总之，界面的定制使用户体验越来越好，给用户带来了亲切感，从而拉近了软件与用户的距离。

10.2 可用性

每个软件使用者对可用性的理解，很大程度上依赖于其知识背景和专业技能，因此，对可用性有一个正式的并且对所有使用群体都适合的定义是不现实的。首先将不同的用户进行分组，然后针对具体类别的用户对可用性进行一般性的定义。

很多标准和规范也是以这样的方式对可用性进行阐述的。在以下章节中，主要考虑广泛使用的 ISO 9241 中的 110 部分。总体上，ISO 9241 规范中针对可用性做了描述，其包含的主要部分和各自的作用如图 10.1 所示。ISO 9241 是关于办公室环境下交互式计算机系统的人体工程学国际标准，由 17 个部分组成。根据人体工程学和可用性原理，分别对各种硬件交互设备属性和软件用户界面设计问题做了详细的规定和建议，并且可以对产品设计符合该标准的程度进行评估和认证。

与其他部分的名字比较起来，图 10.1 中的“部分 110”的名字有些与众不同。ISO 规范一般 5 年修订一次，部分 110 的由来是因为它是早期版本 10 的修订版本。其余 7 个部分没有进行大范围的本质上的修改，只是在旧版本的基础上进行了细化和具体化。

这些子规范在实现中总是相互依赖的，所以，一个实现结果可能会对多个规范产生正面或者负面的影响。另外，依赖于项目类型，不同的项目中可能会侧重不同的子规范。以下是关于 110 对话结构原理的描述和说明。

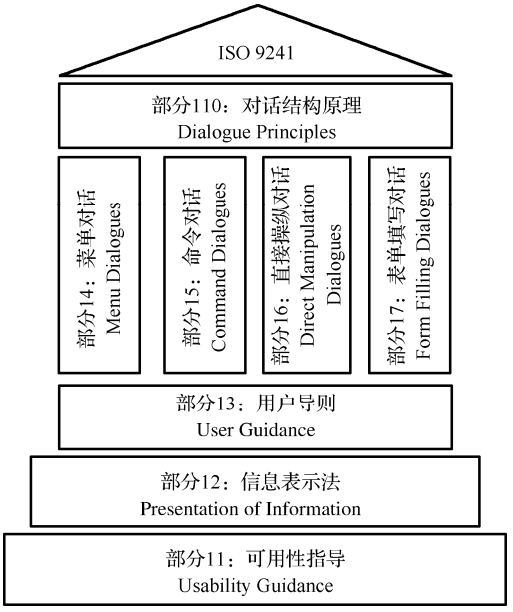


图 10.1　ISO 9241 体系中的软件部分

10.2.1　任务适合性

交互系统在支持用户完成任务时应适合任务。也就是说，功能和对话是基于任务特征的，而不是基于实现任务的技术的。若交互系统能够高效地支持任务的完成，且不需要用户关注界面的特性，则该交互界面是任务适合的。例如，在录入一个新的项目数据时，不需要用户使用鼠标在需要录入的数据项处单击以获取录入的焦点；交互系统可以提供快捷方式迅速地在数据项之间切换，如常用的 Tab 键。

一般来说，任务适合性要求软件能够支持业务的工作流程，为此，每个单一的工作步骤都需要进行详细分析。例如，若要支持项目管理中在项目之间进行差异比较的功能，则应该在工作流中加入多个项目概况的显示列表，并支持项目选择。

另外，任务适合性的重要方面在于界面的复杂程度，也就是界面上显示信息内容的多少。基本的要求是只显示那些本次操作需要的内容，从而降低界面的复杂度。但是在较为复杂的系统中，业务需要显示的信息量依然会使得界面过度“饱和”，这时可以考虑进一步对信息进行合理的编排和构造，如使用标签页的显示形式。

如果在工作步骤中存在一些不需要改变的重复内容，那么对话设计中必须能够支持对这些内容的默认录入。例如，为一个项目添加多项新的任务时，这些任务在录入时不需要重复输入该项目的编号。表 10.1 总结了任务适合性的具体原则和解释。

表 10.1　任务适合性的具体原则和解释

原　　则	解　　释
对话应该为用户提供成功完成任务的相关信息 注意：任务需求决定了所需信息的质量、数量和类型	1．在即将发生的结果与时间非常相关时，对话应显示剩余的时间 2．对话应该提供与情景相关的帮助，如工作步骤

续表

原　则	解　释
避免提供与用户成功完成任务不相关的信息 注意：显示不恰当的信息会导致效率降低，并增加用户的认知负担	不相关的信息应隐藏，如飞机订票系统中应只显示指定日期尚有空余座位的航班，其他航班应隐藏
输入和输出的格式要与任务相符	如日期的格式，中国用“yyyy/mm/dd”，美国用“mm/dd/yyyy”，德国用“dd.mm.yyyy”，界面应能够根据任务种类进行调整
录入的典型值应自动设为默认	(1)飞机订票系统的日期默认为当天 (2)业务的经手人默认为当前登录用户
对话要求的步骤应包括完成任务必须的步骤，省略不必要的步骤 注意：(1)不必要的步骤包括可由系统完成的行为；(2)用户执行常规任务时，对话要帮助用户最小化任务步骤	(1)录入联系人的城市和区号两个输入框，会根据输入的区号自动显示城市 (2)选择对金融行业不感兴趣后，界面应自动屏蔽关于金融行业的相关内容
当一个任务需要源文件时，用户界面应该与源文件的特征兼容	保险公司的纸质文档是计算机的输入源。填入窗口的对话框应与纸质文档的结构一致，无论是元素安排，还是分组内容及输入值的单位等
输入和输出的通道应与任务相符	(1)工业现场由于噪音而不使用声音作为提示 (2)工业现场通过触屏方式进行输入便于执行任务

10.2.2 自我描述性

如果从界面的结构上，就能够清晰地知道什么时间、哪些交互可能发生，为什么发生、会产生哪些可能的结果，那么该界面称为自我描述的界面。相应地，需要给出每个可能步骤的解释，并清楚地说明为什么某个控件无法继续工作。

对话的自我描述性要求：任何时候，对话对用户都能够明显地给出状态提示，如用户目前处于哪段对话，在对话的哪一个阶段，可采取什么操作，操作将如何实现等。

对话过程中一种典型的做法，是将当前未满足条件而无法提供服务的控制置为灰色不可用的状态，使用户意识到当前的操作无法进行。对用户操作的提示的一种具体的实现方式是借助上下文相关的“气泡帮助”机制，一般的工作方式是当用户的鼠标指针指向某个控件时，系统会弹出一个很小的带有一段说明的气泡，解释该控件的作用、目的及使用方法。

为使用户能够快速地熟悉和习惯对话的界面，系统采用的术语应尽量与用户熟悉的业务领域保持一致。若某种操作(如较复杂的计算)耗时过长，系统应提示给出该操作处理的时长和进度。如果用户需要手工录入某个字段的数据，系统应提示数据的格式，或者给出一个示例。同样，也可给出期望的输入格式，如“DD.MM.YY”，使用户清楚要求的输入内容。

在嵌套的菜单控制中，给用户显示出如何到达菜单的层次位置，有时也是非常方便和有用的，一些 Web 页面的导航设计中经常应用，例如，显示出当前页面的位置及其在网站中的层次，如“主页→专业→计算机科学→课程列表”。配合超链接使用，可以清晰地显示出页面所处的位置，方便地浏览网站内容。表 10.2 总结了自我描述性的具体原则和解释。

表 10.2　自我描述性的具体原则和解释

原　　则	解　　释
任何对话步骤所出现的信息应帮助用户完成对话 注意：信息包括向导、反馈、状态信息等	飞机订票系统允许用户输入所需信息并通过“下一步”和“上一步”按钮在对话步骤中导航
在交互中，尽可能避免查阅用户手册或其他文档	通过气泡或底部信息栏提示鼠标指针所指内容的解释
用户应该了解交互的状态	电子商务程序清楚地显示用户完成购买所需的所有步骤及当前所处步骤
当需要输入时，系统应给出格式要求	电子商务程序中，需要用户输入信用卡到期日期时，应显示出期望的格式“mm/yyyy”
对话中用户与系统的交互应清晰易懂	带有图标的命令按钮或菜单项，如保存图标，重做图标，退出图标等
交互系统应说明对格式和单位的规定	燃料类型若为煤，则提示单位为吨；若为天然气，则提示单位为立方米

10.2.3　可控性

对话具有可控性是指用户能够初始化并控制输入的类型及交互过程的走向、步骤和速度，直到达成目标为止。

具体地，若输入的数据不存在彼此依赖关系，则它们的输入顺序不是强制性的。另外，应提供多种方便的交互控制方式，如借助键盘或鼠标等。如果输入过程被中断，那么对话应能从中断处恢复并完成余下的处理，已经录入的数据不需要重新录入。可控性的具体原则解释见表 10.3。

表 10.3　可控性的具体原则和解释

原　　则	解　　释
用户交互不应被交互系统完全决定，应在用户的控制之下，由用户根据自己的需求和特点进行调整 注意：(1)特定的交互系统可能会包含明显的要求，在特定情况下不允许用户控制，如在测试环境下，时间约束也是测试的一部分。(2)在一些场景下，任务本身决定了交互步骤的烦琐程度。(3)用户交互步骤可能会受到业务限制，如“空闲 2 小时后离线”	手机允许用户编辑短信，显示出已经录入的字符数，直到用户决定发送、存储或删除该短信为止(不管用户写完这条短信要花多长时间) 教师在教务系统录入考试成绩时，应有足够的时间进行录入，不会受到系统对话期限的限制
用户应控制如何继续对话	当在账务系统中登记一笔支付时，系统自动选中最早的未付订单，但也允许用户选择其他订单来支付
如果对话被打断，用户应该能够决定从何处重新开始，如被打断之处	在线视频播放应允许用户选择是从上次停止处继续观看还是重头观看

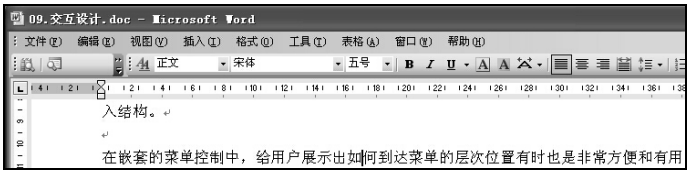
10.2.4　与用户期望一致性

与用户期望一致性是指对话行为与用户的期望相符，用户的期望来自用户对其他交互界面的经验及用户的业务领域，与任务适合性具有清晰的联系。对话如果与用户可预见的场景需求及普遍沿用的管理保持一致，则称为具有“与用户期望一致性”。

首先，交互界面在相同的条件下应该具有相同的行为，如错误提示都在屏幕中间弹出的窗口中进行显示，系统的当前状态都在窗口下部的状态栏中进行提示。以比较常见的办公软

件为例，其主要功能就是做文字处理，在窗口的最上部分通常都是一个将各种任务以层次结构组织起来的菜单。

作为一种时尚，微软的 Office 界面在较新的版本中将菜单项的形式由传统的菜单控制转变为选项卡的风格 Ribbon，通过将每个任务域中所有重要的功能显示在窗口的上部，提供更为直接的选择。图 10.2(a)显示的是传统的 Word 2003 中的窗口效果，其中的工具条是可以配置的，图 10.2(b)是 Word 2013 选项卡风格的窗口效果。



(a) 传统风格



(b) Ribbon 风格

图 10.2　两种风格的 Office 界面

另外，与用户期望一致性的要求是交互系统能够快速提示用户是否可以录入，录入的数据是否合理正确。例如，在一个聊天程序中，聊天室的内容显示背景与输入聊天内容的背景应不同，以区分允许或禁止输入。与用户期望一致性的具体原则和解释见表 10.4。

表 10.4　与用户期望一致性的具体原则和解释

原　　则	解　　释
应使用用户熟悉的词汇，基于用户已有的知识	财务系统应采用专用的术语，如“借方”“贷方”等
应给予用户及时、恰当的反馈，符合用户期望	安装软件成功时，用户能得到安装成功的提示
如果反应时间与用户期待的时间差距较大，应告知用户	当用户进行复杂的计算处理时，系统提示计算的进度或者处理的速度
数据结构和组织表格应尽量以用户觉得自然的方式显示	在线商店以近似于商店里实际布局的方式来显示产品
格式应遵循正确的文化和语言惯例	文字的显示对英文采用左排序，对阿拉伯文采用右排序
反馈和说明的长度应基于用户的需求	根据用户实际的需要选择反馈的长度和方式，如图纸设计系统中，除了显示最终的工业图纸，还要给出一份计算说明书
一个交互系统中贯穿任务始终或是相似任务的对话行为和外观应保持一致	(1) 软件界面风格、按钮位置等应一致 (2) 数据备份与还原界面应采用相似设计
如果基于用户期望的一个特定输入之处是可以预期的，那么当对话要求输入时，该处应为输入做好准备	在软件包安装过程中，每一步要求用户动作的对话均可由按键盘上的“回车”键来实现
给用户的反馈或信息应规范化，并以客观的风格显示 注：特定的应用领域例外，如娱乐软件会采用更新奇的风格	如果用户输入的日期不符合规范，错误信息为“请输入 dd/mm/yyyy 格式的日期”

10.2.5 容错性

对话的容错性是指即使有错误的输入，在系统错误及其类型的提示下，只要进行很少的修改，就能够得到正确的工作结果。容错性对于对话来说，最基本的要求是，错误的输入不会导致数据的丢失或程序的崩溃。

容错性要求软件系统能够检测出用户操作关键步骤中的一些非典型情况。一个常见的例子是用户在关闭系统之前，提示用户是否要对尚未保存的数据进行保存。另外，对用户录入的数据要能够进行实时验证和提示，以保证数据的正确性。例如，在一个要求填入数字的输入域，如果填写了其他符号，可以将输入域的背景颜色置为红色，以示提示。

容错性还要求交互系统具有对错误上下文敏感的帮助系统，如果可能，这个帮助系统能够指示成功完成该项任务的步骤和条件。错误提示要求具有较好的可读性和建设性。另外，交互系统的容错性也应允许用户在某种程度上对任务的执行进行尝试，并且能够返回到尝试前的状态。

10.2.6 可定制性

如果交互系统具有根据不同用户的能力和喜好进行设置的能力，则称交互系统具有可定制性。许多软件在选项设置中提供了很多用户自定义的功能，如显示字体的大小、颜色等。除此之外，提供对显示颜色的配置能力以适应不同场景的需要，也非常有实际意义，如通过投影仪演示的场景或者要求高对比度的特殊场景等。

可定制性同时也体现出已经介绍的可控性原则，如用户可以将多个工作步骤定制合成到一个大的步骤中完成。另外，自我描述性中的气泡式帮助虽然有用，但对于有经验的用户来说，如果系统经常给出这样的提示，反而会对工作产生干扰，因此可定制性要求交互系统能够提供开启和关闭该提示的设置。总之，交互系统的可定制性提供给用户对工作风格和习惯等完全可定制的服务。

10.2.7 易学性

对话的易学性意味着对话应支持和指导用户学习使用该系统。在前面提到的一些原则也属于该范畴的要求，交互系统重要的是要使用用户的业务术语并且提供用户认识和学习该系统的机会。例如，允许用户在对话中尝试可能的工作步骤并能够回到之前的状态。另外，与易学性直接相关的还有系统提供的文档及附加的培训材料等。系统在可能情况下应提供一些具体的实例，用户可根据这些实例举一反三，产生相似的解决方案。例如，在项目管理的例子中，系统可默认提供一个虚拟的项目，它可以用于完成下载和实验等学习目的，并且可以作为用户使用手册中的基本示例。

易学性还要兼顾用户中不同的学习群体，有的用户善于通过具体的示例学习，有的用户习惯于导航式的 Wizard 对话学习。易学性的具体原则和解释见表 10.5。

表 10.5　易学性的具体原则和解释

原　　则	解　　释
应该展现给用户“有助于学习的规则和隐含的概念” 注意：有助于用户形成理解业务的模式和策略	一个压缩文件的软件应该解释清楚“文档(Archive)”的概念
对非专业用户或是不常见的业务，应提供适当的帮助	财务软件应提供引导用户按规定走完要求的对话步骤(Wizard)，以创建年度资产负债表
提供适当的支持，帮助用户熟悉系统 注意：不同的用户对帮助有着不同的需求	当用户按下指定的帮助键时，软件应能解释每个不同菜单项的使用
反馈或解释应该有助于用户对整个交互系统形成概念性的认识 注意：对话最好能适合不同经验的用户需求	如 Office 系统中的所见即所得(WYSIWYG)
对话应提供对任务中间结果和最终结果足够的反馈，以便用户能从成功完成的任务中学习业务	用户使用旅馆预订系统预订房间时，用户可接收到每一步的反馈，提供改善查询和成功预订的细节
如果不妨碍正式的业务，交互系统应允许用户探索对话步骤，而不会受到负面影响 注意：可根据实际情况决定是否支持	生产计划排产应提供预排产功能，针对不同的资源优先级尝试不同的排产方案
交互系统应在业务过程中做到让用户需要输入和了解的内容最少化	生产计划排产应提供“一键排产”按钮，减轻计划人员的工作负担，并可在结果上进行手动微调

10.3　交互设计过程

界面交互设计不仅对项目结果的接受程度具有较大的影响，而且对整个软件开发过程同样具有影响。需求分析的主要目标就是识别和了解哪些业务需要在待开发的软件中进行实现。这个阶段确定下来的在软件中支持的业务流程同样也是进行界面设计的根据。因为针对主要功能的设计是面向最终用户的，所以，在设计的过程中应充分考虑如何尽量减少用户的操作步骤，以完成工作。

需求分析的另外一项工作做数据的建模，即了解主要业务步骤中都需要哪些信息。这个工作一方面理清数据之间的逻辑关系，为业务功能做好准备；另一方面，通过对数据流的分析，了解如何对数据的处理过程进行分解。

另外一项需要分析阶段的目标是决定典型终端用户具有的特征。这可以通过对有经验的用户进行问卷调查，询问哪些条件需要在业务操作中进行考虑。例如，确定未来系统的使用地点，尤其是在一些工业现场，计算机屏幕可能很容易变脏或者用户需要戴手套进行操作，还有可能现场噪音较大，用户无法听到系统默认的提示音等。这些都需要在需求分析中进行识别和考虑，除了在硬件的选择上要符合要求，在软件的交互设计上同样需要满足实际操作的需要。

如果是对已有软件系统的重新开发，那么就需要考虑如何将旧系统中用户已有的工作流程在新的系统中进行迁移。一个简单的做法是，以原系统的界面为基础，熟悉并保持已有业务的实现方式，并做进一步的扩展。另外，在软件的界面设计中，尽量在架构和设计原则的层面上对交互进行抽象，使实际的核心业务功能与界面间的接口保持独立，最小化在交互中进行的数据交换和管理。

在界面交互设计过程中，初步的设计工作可以绘制在纸上或者白板上，以方便考虑每个

业务步骤粗略的图形界面及其布局，类似动画制作中的故事板，为每个工作步骤勾勒大概的界面轮廓，并将各步骤使用箭头进行连接。然后对这些界面进行原型化的实现，可以借助一些图形界面设计的辅助工具，它们支持方便的控件拖动等可视化的设计方式。界面的设计原型需要与客户一起讨论并不断优化，这是一个非常重要的过程。客户不懂技术，开发者不懂业务，可视化的界面原型能够在两者之间搭建起沟通的桥梁。同时，客户也可以在不同程度上参与技术实现。另外，这个沟通的过程也可以提升开发者对业务的理解，客户在初始界面原型的基础上，思路也可能会进一步拓展，从而提出更富有建设性的功能性需求。界面的交互设计过程拉近了客户与开发者的距离，同时也是一个风险释放的过程。

10.4 可用性的验证

在软件开发中，一个核心的任务就是确保期望的功能在未来的系统中得以实现。验证软件的功能实现主要在软件测试部分进行说明。交互设计的测试需要在软件功能开发后进行，但一般会采用不同的测试方法，因为交互设计的测试通常不能简单、独立地自动运行。本节主要针对交互设计的验证方法进行说明。

对交互设计的测试，一般采用两种方法。一种是基于领域专家的方法，主要依赖专家的经验进行评估；另一种是基于最终用户的方法，通过跟踪和调查最终用户对系统的使用情况进行分析。交互设计的测试要围绕可用性的要求，重点考查系统对最终用户的业务操作流程是否提供了最优的支持，包括以下内容。

(1) 界面整体上是否具有统一的设计？用于界面开发的软件或软件包的选型是否合适？

(2) 目标系统是否能够给用户提供一种循序渐进的使用体验？是否为用户提供了规范化的业务指导作用？

(3) 用户重点强调的特征是否在交互系统中得到了贯彻和实现？

具体地，可以从以下 5 个基本方面进行验证，并可根据项目的具体情况进行调整和组合定制。

1. 启发式评估

这是基于领域专家的方法。专家应该在相应的领域具有较好的专业知识和经验，并且对交互设计未来的发展趋势具有较好的前瞻性。具体的做法一般是将多个专家按照不同的评估方面进行分组，分别评估，然后将他们的评估结果汇总，形成一份评估报告。这种方法的好处是能够利用专家的经验给出比较中立的评估结果，但挑选并且聘请专家进行评估的过程需要较高的投入。

2. 基准和检查表

基于领域专家的方法，评估对象一般是针对系统的规格说明文档，我们还可以在此基础上设计和利用与可用性相关的检查表进行辅助评估。

表 10.6 是对项目管理系统设计的一份评估检查表，表中给出了评估项的列表和每个评估项对应的权重系数。评价结果的范围为 0～4，0 表示不满足，4 表示非常满足。评估可由不同的人分别完成，他们可以是精心挑选出来的不同涉众的代表，如没有参与界面设计的开发者、

其他项目的同事、内部专家、客户代表等。评审的结果会被收集和分析，并为界面设计的下一次迭代指明方向。

表 10.6　交互评估检查表示例

序　　号	评估项	权重系数 g 重要程度(4→1)	评价结果 b 满足程度(4→0)	综合值 g*b
1	项目便于添加	3	4	12
2	项目便于比较	4	2	8
3	进度估算容易进行	3	3	9
4	估算结果容易理解	4	4	16
5	统一的系统错误提示	2	2	4
合计(共 64)		—	—	49

这种方法的好处在于给出的评估标准清晰、明了，可以对评估结果做出细致的分析。不足之处在于，该方法需要针对每个项目对评估标准进行调整，尤其是每个评估标准中的评估项和权重系数的选择是比较困难的，要兼顾全面性和有效性，并力求发现所有潜在的交互问题。

3. 用户调查

这是一种基于最终用户的方法，通常提供给用户一份调查问卷，形式上多是一些客观选择题，内容类似于检查表的形式，也可以补充少部分自由回答的问题。

除了可用性的问题，被调查者的人机交互知识、对交互系统的期望及改进建议等也可以列入相应的调查问题。总之，要求问卷的创建者要至少具备一些社会调查方面的经验，不仅能够对问卷的结构进行设计，而且能够对调查结果进行分析，以确定结果的真实影响。

4. 基于任务的测试

这是一种基于最终用户的方法。为此，需要选择一组系统最终用户，他们可以是实际的客户，也可以是其他具有业务领域知识的人。在测试的过程中，这些被测试人员将被分配某些业务任务，并通过目标系统来完成。每个人完成任务时使用软件的情况将会通过屏幕录制或其他方式保存下来，如鼠标和键盘的使用。随后，开发者将会通过这些记录进行用户行为分析，以寻找有潜在优化机会的地方。还可以与被测试人员在一起座谈，了解他们对新系统的印象和感觉，为交互设计的改进提供直接的信息。

实践经验表明，基于最终用户的测试方式只需要较少的测试人员，就可以得到需要的结果，根据系统的复杂程度，一般可以安排 4～10 人进行测试。该方法存在的一个问题是，相应的被测试人员难以寻找，尤其是对业务较熟悉但不需要太多培训就能参与的人员。

5. 放声思考的测试(Thinking Aloud Test)

这是“基于任务的测试”的一个变种。此方法除了要求记录每个被测试人员的行为，还需要他们说出和解释所做的每个步骤的确切想法。该方法最大的好处在于，除了能够观察被测试人员的动作，还能够洞察他们的工作方式。这一切能够直接告诉开发者用户在寻找什么，对改进对话界面或者功能的结构是非常有价值的。

该方法的一个主要问题也是合适被测试人员的寻找。除了需要具有最终用户的业务经验，被测试人员必须能够准确描述他们的想法，并且以一种他们习惯的方式完成测试任务，也就是要求他们要有头脑，知道每一步操作的目的和想法。

10.5 习题

1．本章描述的 ISO 9241 对话原则适用于所有软件系统中的界面设计。请针对每个对话原则，根据自己对某软件系统(如腾讯 QQ、新浪微博、淘宝等)的使用体验，分别给出 2 个正面的应用示例并简单描述。

2．扁平化设计(Flat Design)就是在进行界面设计的过程中，去除所有具有三维突出效果的风格和属性，即去除下落式阴影、梯度变化、表面质地差别，以及所有具有三维效果的设计风格。很多流行的系统都采用了扁平化的设计方式，如 IOS、Android 和微软 Metro 风格的系统。请结合交互设计的原则，解释扁平化界面设计的优势和不足。

第11章 软 件 测 试

质量保证是软件工程中一个较为宽泛的领域，具体又可以细分为产品保证和过程保证两个方面，分别在产品和过程两个层面提供一整套有计划、有系统的方法确保软件的质量。过程保证的相关内容，将在第 13 章中进行介绍，本章主要围绕产品保证中的软件测试展开，讨论如何确保开发的软件满足用户需求。

首先介绍形式化验证方法，简要说明对系统正确性进行验证的理论和技术。接下来，对测试技术进行概述，包括测试分类、测试策略等。除测试外，质量保证的活动还包括对质量影响程度的度量，也就是软件度量，它对开发过程的支持和改进也会起到非常重要的作用。常见的度量指标和计算方法也会在这部分进行介绍。

对于系统功能实现，软件工程领域中已经提供了较为系统化的测试方法。在测试设计中，重点讨论两种有代表性的测试用例设计方法，分别是等价类分析和基于控制流的测试。测试用例设计的一个基本原则就是，测试内容的选择既要简短，又要能够尽可能多地发现缺陷。

断言机制和测试框架是两种基本的测试实现技术，它们在编码的层次上对系统正确性的验证提供了基础性的支持。

最后介绍可测试性方面的内容，即如何进行功能实现，以提高系统的易测性，分别从测试程序和被测程序的两个方面进行说明。

11.1 形式化验证

软件测试除穷举外，还存在一些系统化的方法，能够大大提高软件测试的效率及软件质量。软件质量即软件的正确性，它是用户需求的具体体现。如何对正确性进行评价？参照又在哪里？软件测试给出的基本原理是，用户需求是否被准确表示并实现。因此，对于一个软件系统来说，我们可以事后对其进行验证，验证其是否达到了用户标准。为此可以根据不同的需求，准备一系列的测试，让其以某种形式运行起来，以实现对系统的验证。注意，测试的顺利通过并不能用来证明整个系统是正确的，因为测试数量通常是有限的，测试内容的选择策略通常是对系统可能的缺陷进行分类，并有针对性地让其表现出来，这并不能代表全部可能的情况。

对于正确性的验证，一种直接的方法就是编写一个验证程序，将待测程序和需求传入，然后该程序给出需求是否被满足的结果。但是，可计算性理论对停机问题进行了研究，给出的结果证明这样的程序是不存在的。也就是说，不可能用一个单独的程序来判定任意程序的执行是否终止。

若能够限制需求的类型，则验证过程是可以进行的。这样的思想实际上是建立在模型检测(Model Checking)的基础上，通过一个确认程序对模型的语言和需求描述同时进行处理，可实现自动对模型是否满足某个具体的需求进行判断。模型检测的基本思想是用状态迁移系统

(S)表示系统的行为，用模态逻辑公式(F)描述系统的性质。这样“系统是否具有所期望的性质”就转换为数学问题“状态迁移系统 S 是否是公式 F 的一个模型？”。对有穷状态系统，这个问题是可判定的，即可以用计算机程序在有限时间内确定。模型应使用某种模型语言，实现对目标系统尽可能简单的表示。

基于这样的思想，若给出若干有限的逻辑条件和结果，则可以对模型表示的所有可能情况进行检测。这里实现了一个返回类型为布尔类型的方法，其包含 if-then 的逻辑形式，对应的具体需求见表 11.1。根据此需求实现的具体程序如代码 11.1 所示。

表 11.1　逻辑条件与结果的对应需求

a	b	逻辑结果
true	true	true
true	false	false
false	true	true
false	false	true

代码 11.1　逻辑检测的实现

```
public static boolean implementLogic(boolean a, boolean b){
    return !a||b;
}
public static void main(String[] args){
    boolean values[]={true,false};
    for (boolean a:values)
    for(boolean b:values)
        out.println("a=" + a + " b = " + b + " result: "+implementLogic(a,b));
}
```

现在可以对所有可能的情况进行检测，对照表 11.1 中说明，验证是否满足需求。程序的输出如下：

```
a=true b=true result: true
a=true b=false result: false
a=false b=true result: true
a=false b=false result: true
```

这个例子也同时说明一个准确的需求描述对于测试的重要性。若想对任意的编程语言和逻辑进行正确性的证明，则需要使用证明系统。其主要思想为：通过对较小的已经被证明的程序的使用及其证明的规则推导出较大的正确的程序。需求在这里作为程序的一种性质以逻辑的方式体现。程序 P1 在具有性质 E 及满足逻辑 B 的情况下，可实现性质 F 的要求；程序 P2 在满足性质 E 及 not B 的逻辑下，也实现了性质 F 的要求。然后就可以通过证明的规则得到以下隐式的结论：如果性质 E 得到满足，通过实现可得到规则 if (B)then P1 else P2 满足性质 F 的要求。当然，实际程序的证明和推理步骤是比较烦琐的，其核心是一个定理证明系统。

模型检测的方法在 20 世纪 70 年代末被人们应用在计算机硬件系统的验证中，因为硬件相对于软件来说，不容易改变。模型检测方法的高效应用需要专家的参与和研究，是否可以或者如何在传统的软件开发中进行应用，还是一个开放的问题。前面章节中介绍的对象约束语言(OCL)其实可以作为一个很好的切入点，因为它允许在类模型中融入对需求的表达，这是证明系统可以利用和处理的内容。

11.2 测试技术

软件测试(Software Testing)的定义是：使用人工或自动的手段来运行或测定某个软件系统的过程，其目的在于检验它是否满足规定的需求或弄清预期结果与实际结果之间的差别[①]。软件测试通常被认为只是开发后期的一次性活动，但实际上，应该与整个开发流程紧密地融为一体。敏捷思想的普及使软件测试得到了前所未有的重视，它也是项目成功必不可少的基石。

另外一个与测试相关的概念是测试用例(Test Case)，测试用例是为某个特殊的测试目标而编制的一组测试输入、执行条件及预期结果，以便测试某个程序路径或核实软件是否满足某个特定需求。测试用例其实就是测试的具体内容，也是软件测试设计的主要工作。

11.2.1 测试分类

软件测试可按照图 11.1 所示的 V 模型进行阶段划分，在每个开发的迭代周期中，需要从上至下实施所有测试阶段。第一阶段为单元测试(类测试)，主要对每个单一的新开发的功能模块进行测试，并且多由开发者本人按照质量保证相关规范(如要求对类中的每个方法都进行一次独立的测试，并且代码的覆盖率至少要达到 95%等)进行实施。第二个阶段为集成测试，对经过单元测试的类逐步进行集成，构成最终的包或者系统。此过程重点关注每个类与其他相关类的交互，经过集成测试的包会被进一步集成为完整的系统或构件，最后这些系统或构件还需要与和其他预定义的接口进行通信的系统一起，进行进一步测试，从而构成最终完整的应用系统，这是第三个阶段，即系统测试的主要任务。上述的测试阶段一般都是由软件开发者进行的，最后一个阶段是验收测试，一般由客户主导，主要是将系统测试中的工作在用户现场重复执行，并在此基础上加入一些客户自己的测试愿望，如采用完全真实的数据进行测试。

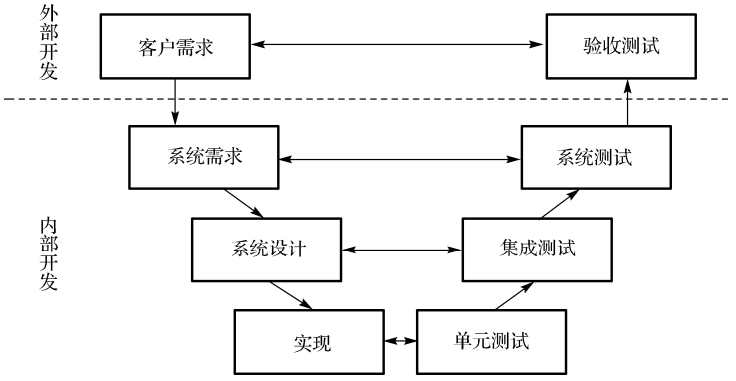


图 11.1 测试阶段

另外，还可以根据被测对象被关注的细节程度的不同，将测试分为白盒测试、灰盒测试和黑盒测试，如图 11.2 所示。白盒测试关注的是被测对象的内部构成细节，如算法的结构和流程，所以多在单元测试阶段采用。灰盒测试一般在集成测试阶段中使用，因为这个过程关

① 来自 IEEE 提出的软件工程术语标准。

注的是类、包等程序单元之间的关系，而不是类内部的细节。系统或验收测试阶段一般使用黑盒测试方法，此时系统内部的细节不再重要，重要的是系统的外部行为。

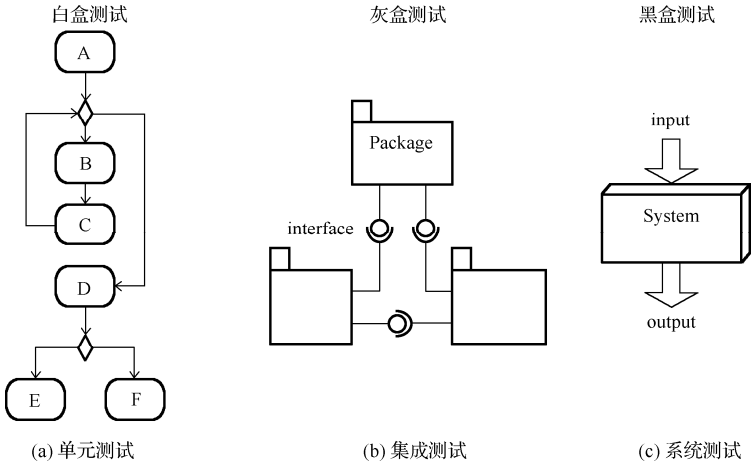


图 11.2　测试的粒度

11.2.2　测试策略

测试的阶段与采用的测试方法是相互配合、互为支持的关系。如果将所有的测试都在系统测试阶段进行，那么由于系统测试关注的是系统整体的功能和行为，当某个缺陷被发现时，需要花费大量的成本来定位缺陷。缺陷修正的成本曲线如图 11.3 所示，可以看出一是缺陷修正成本与缺陷的类型有关，二是缺陷越早被识别并处理，缺陷修正成本也越低。

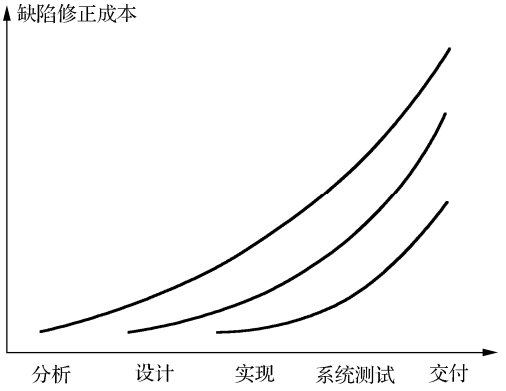


图 11.3　缺陷修正的成本曲线

图 11.4 中给出了不同开发阶段中的 UML 模型和测试阶段之间在总体上的对应关系，其中的实线表示支持，虚线表示辅助支持。当然，这些对应关系不是绝对的，但应该在开发的早期就考虑到这些分析设计模型对后续的可测试性及测试方法的支持。

在实际的开发中，所有的测试用例都应该集中进行管理，每个测试用例都要有具体指定执行的条件及预期的结果。这在增量式开发过程中非常必要，因为上一个迭代周期中的测试用例需要在本次的迭代周期的测试中重新执行，以确保没有引入新的缺陷，这称为回归测试。测试用例的管理过程如图 11.5 所示，每次迭代在测试用例库中都会加入新的测试用例，并且要求其中所有的测试都要成功通过。

测试迭代过程如图 11.6 所示，每次迭代的测试用例数量都在逐渐增加，并且需要新旧测试用例共同执行，完成本轮测试。当然，这个过程比较烦琐，尤其是在大型的项目中，通常会借助一些自动化管理工具辅助进行，如持续集成工具 Jenkins 等。

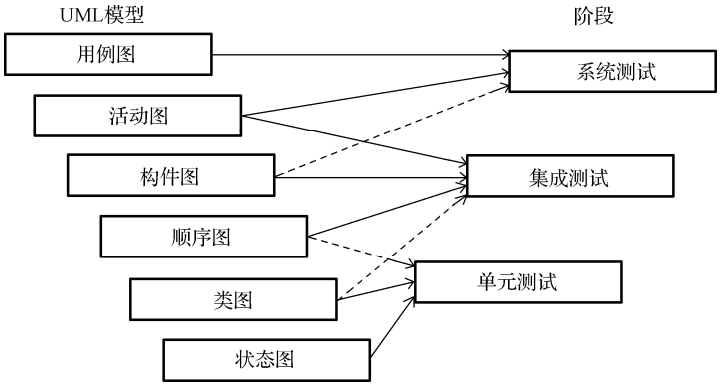


图 11.4　UML 模型与测试阶段

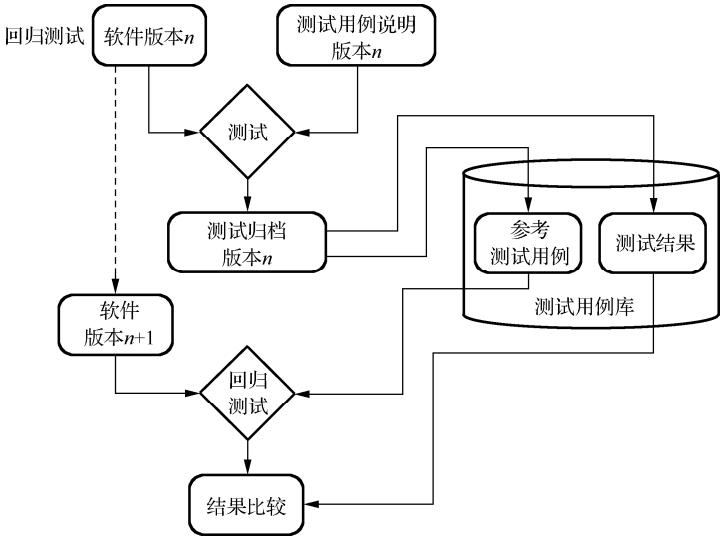


图 11.5　测试用例的管理

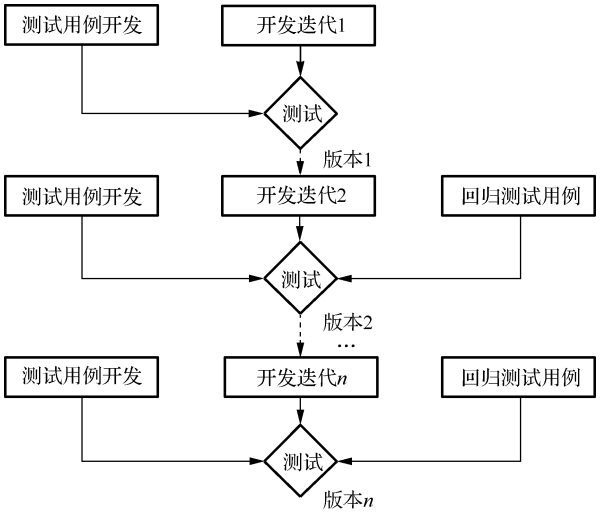


图 11.6　测试迭代过程

11.2.3 非功能性测试

除了功能性测试，对软件系统的测试还包括非功能性测试，如性能测试、安全测试、安装测试、配置测试、界面测试、容量测试等，都是其常见的形式。

性能测试是检查系统是否满足需求规格说明书中规定的性能，检验相关指标能否达标、是否能够保持。性能测试需要模拟实际用户负载来测试系统，包括反应速度、最大用户数、系统最优配置、软硬件性能、处理精度等。性能测试一般结合压力测试、负载测试等手段，在大访问量时尤其需要使用工具来进行。压力测试主要用来获取系统能正常运行的极限状态，一般做法是在人为设置的系统资源紧缺的情况下，检查系统是否有功能或者性能上的问题；负载测试是通过改变系统负载方式、增加负载等方式发现系统中所存在的性能问题，以检验系统的行为和特性。图 11.7 中描述了不同的性能测试种类及其特征，横坐标为测试持续的时间，纵坐标为系统的负载。

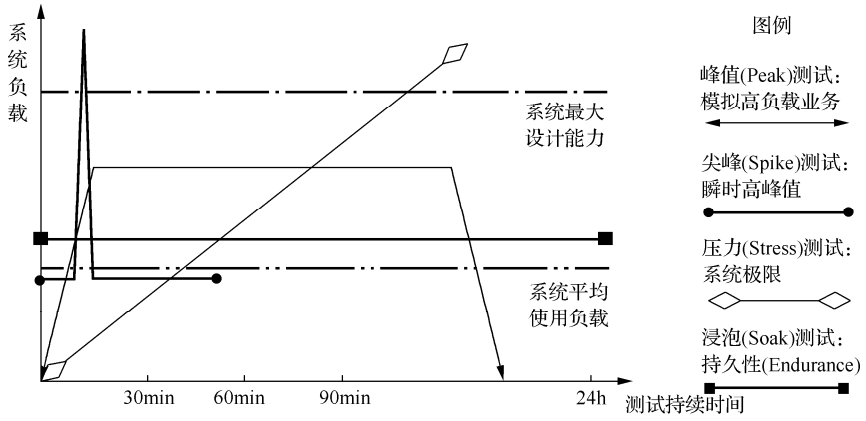


图 11.7 性能测试的种类

界面也是系统的构成部分，同样需要保证开发和使用符合要求，因此也需要进行测试。界面的交互设计和验证方法已经在第 10 章中进行了单独的介绍。系统级的测试一般会涉及界面交互，比较适合使用等价类的方法来建立对应的测试用例。这样的界面测试首先面对的一个技术问题是，标准的测试框架通常无法直接执行界面的交互。为此需要使用支持“捕捉和回放”(Capture and Replay)的工具，辅助记录界面使用时的详细动作，如输入的数据及鼠标的拖动和单击等事件，包括输出结果的记录，如结果文本或提示颜色等。测试用例的生成需要使用该工具对被测试界面通过一组标准操作进行录制，然后在每个新的界面发布后对用例脚本进行回放，从而实现测试的自动化执行。常用的此类工具如 JMeter、Selenium 等，它们支持的脚本类型及侧重点各不相同，需要根据实际的需要选择使用。

11.3 软件度量

第 10 章为了评估界面的质量，建立了一个交互评估检查表(表 10.6)，其中为每个参照标准界定了权重。对交互界面评估时，按照实际情况对照该标准可以计算出一个具体的综合值，

代表对质量的一种度量。在企业管理中，类似的对质量的量化管理是很常见的。

量化的度量指标可以用来客观地评估项目总体的完成情况，图 11.8 给出了针对测试的量化描述图示，从不同的方面对系统测试完成度进行评价。每个柱形上方的横线表示项目具体的目标，通过这种量化的方法可以对项目的状态一目了然。

对产品质量度量的一般方法是：先使用某种规则建立一个指标体系，然后再进行质量评估。这种方法应用到代码上的优势在于，大多数的度量指标可以自动计算，其重点在于指标体系本身的建立和优化。如果建立的指标体系有缺陷，将导致评估结果会对开发者造成误导，从而对项目开发产生不利的影响。例如，简单地追求注释行数和代码行数的比例关系，可能会导致代码的精简度和可读性下降，因为加入冗余的注释虽然可以直接提高该指标值，但对可理解性并没有贡献。

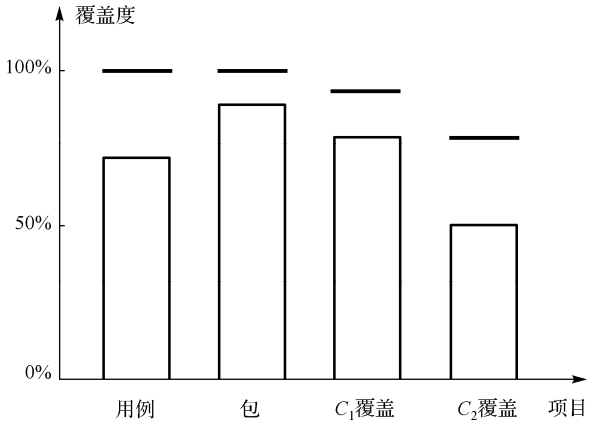


图 11.8　测试用例的覆盖度量

尽管如此，度量是确定软件质量的一种有价值的辅助手段。下面介绍几种不同的度量方法，之前提到的项目规模等指标可以作为简单的基础性质量指标，包括利用注释行数和代码行数的比例来反映注释强度，利用统计方法计算代码行数与方法数的比值确定方法的平均长度等。还存在一些其他的指标，它们作用于不同的程序粒度，并且具有一定的参考价值，可以适当地用它们对产品或者开发过程进行定量的度量。例如：

(1)保持变量和方法名适当的长度在某种程度上可提高程序的可读性。

(2)方法中参数个数反映了方法的复杂程度。

(3)类中实例变量的个数决定了该类信息的丰富程度。

(4)继承的深度提供了对继承使用是否恰当的参考，如过多的继承对应深度的增加，并使重用变得困难。

所有的度量指标可以赋予一个软边界和一个硬边界。指标的度量在必要的情况下可以超出软边界的要求，但硬边界在任何情况下都不允许超出。度量边界的确定与具体的项目相关，项目开发过程中，由于很多影响因素的不同，会造成各种度量指标的差异，如利用图形类库的界面开发，往往继承深度要大一些，可根据情况适当放宽限制。

除了介绍的这些比较通用的统计指标，还有一些针对类及方法复杂度的度量指标，也可以用于质量方面的参考，下面介绍两个常用的指标。

11.3.1 控制流图与环形复杂度

方法的复杂程度可以通过 McCabe 指标进行度量，也称为 McCabe 环形复杂度，它主要以方法的控制流图结构为基础进行计算。

每个类的方法都可以使用一个有向控制流图对该程序段可能的执行方式进行描述。控制流图中的节点表示代码指令，节点间通过有向直线进行连接，表示这些指令执行的先后顺序。if、switch 和循环指令会在图中对应节点处产生分支。

图 11.9(a)为一个方法的代码，其功能是对于所有输入的正偶数输出其平方值，而对于其他输入，输出 0 值。图 11.9(b)为该方法对应的控制流图。控制流图在绘制的时候具有一定的自由度，例如，表示代码段结束的花括号可以作为一个单独的汇聚节点处理或者忽略，再如，对于指令 3 中已经将 i=i-1 的作用蕴含在 i--的形式中，是否需要将其提取出来并作为单独的节点进行描述也是一个问题。因此，为尽量统一地描述程序控制流图，如果指令对应的节点以 $k_1 \rightarrow k_2 \rightarrow, \cdots, \rightarrow k_n$ 顺序出现，以下情况将它们合并为一个节点进行处理：

(1)此序列的执行每次都是从 k_1 开始，除此之外没有边终止于 $k_2, \cdots, k_n$；

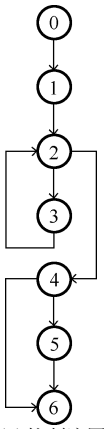
(2)此序列的执行每次都是以 k_n 结尾，除此之外没有边始于 $k_1, \cdots, k_{n-1}$；

(3)满足(1)和(2)的最长节点序列。

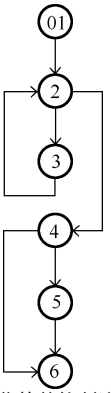
在上面的示例中，按照以上原则，节点 0 和 1 是可以合并的，其余节点不能合并。例如，节点 2 和 3 不能合并，因为从 2 到 4 存在一条边，不符合条件(2)的要求。现在对于是否需要为 i--设置单独的节点也有答案了，因为如果为其单独设置节点，也会由于上述原则导致与前面的节点再次合并。所以，控制流图经过化简后如图 11.9(c)所示。

```
public int test (int input){
    int result = -input/2;                  //0
    int i=input;                            //1
    while (i>0){                            //2
        result = result + (i--);            //3
    }
    if(input < 0 || input%2 == 1){          //4
        result = 0;                         //5
    }
    return result*2;                        //6
}
```

(a)源程序



(b)控制流图



(c)化简的控制流图

图 11.9　控制流图

McCabe 的计算方法为：$V(G)$=边数-节点数+2。图 11.10 中给出了几种基础的控制结构和对应的 McCabe 值。可以看出，控制结构中分支或者循环越多，McCabe 值越大，这也是程序可读性的一个反映指标。公式中的“+2”的主要作用是对 McCabe 值的归一化，以保证其最小值为 1。在计算环形复杂度时需要注意，如果判断语句中含有多个原子谓词组合成的复合条件，那么需要将复合条件拆分成多个判定，并保证每个判定中只含有一个原子谓词，例如，

计算图 11.9 中的程序的环形复杂度时，需要将控制流图转换成图 11.11 所示的形式，其中，边数为 9，节点数为 7，环形复杂度 $V(G)=9-7+2=4$。

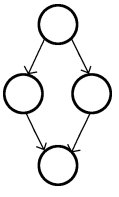
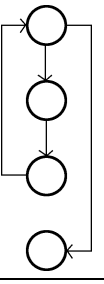
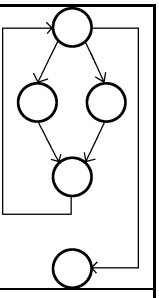
控制结构					
边数	0	2	4	3	6
节点数	1	3	4	3	5
McCabe 值	1	1	2	2	3

图 11.10　McCabe 复杂度计算示例

为降低方法的 McCabe 值，开发者通常需要在开发过程中采用以下几种方式对结构进行改进，即对代码进行重构。

(1)优化方法中算法的结构，使其尽可能简单。

(2)对方法中 if 嵌套的结构进行分解，将外层的 if 语句包含的程序部分移到另外一个局部方法中，这种方法在重构的介绍中已经进行了说明，而且有些工具也能提供相应的支持。

(3)利用多态性，使对分支的选择不再受局部判断逻辑的控制，而是根据程序运行时的实际情况进行动态选择。

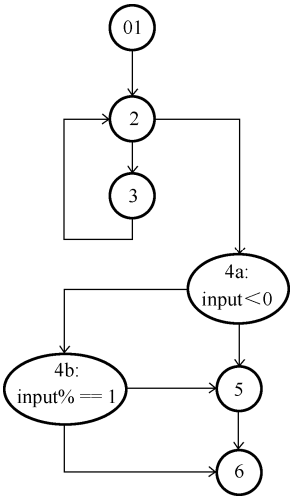


图 11.11　拆分复合条件

考虑到复合条件的情况，McCabe 方法的计算实际上反映了方法中下列语句产生的分支结构：if 语句、条件组合(&&和||)、for 语句和 while 语句。如果计算出的 McCade 值大于 10，就应该考虑将该方法按照上述方式进行改进，并降低复杂度，而面向对象的程序设计中，类方法的 McCabe 值一般限制在 5 以下比较合适。程序结构越复杂，越难于测试和理解，复杂的逻辑会使方法的可理解性降低。McCade 复杂度的计算过程也可简化为以下算法，以便于进行工具的实现：

(1)每个代码段的初始复杂度为 1。

(2)遇到每个原子条件加 1。

(3)每个 switch 中的 case 段加 1。

代码 11.2 给出一个使用简化算法进行 McCade 复杂度计算的示例。

代码 11.2　简化的 McCabe 复杂度计算示例

```
// V(G)= 1
// +2 conditions, V(G)= 3:
if ((i > 13)|| (i < 15)){
    System.out.println("Hello, there!");
```

```
    // +3 conditions, V(G)= 6:
    while ((i > 0)|| ((i > 100)&& (i < 999))){
    //...
    }
}
// +1 condition, V(G)= 7
i = (i==10)? 0:1;
switch(a){
    case 1: // +1, V(G)=8
       break;
    case 2: // +1, V(G)=9
    case 3: // +1, V(G)=10
       break;
    default:
       throw new RuntimeException("a = " + a);
}
```

11.3.2 方法内聚缺乏度

在面向对象的领域中，还存在另外一种度量 LCOM*（Lack of Cohesion in Methods），用来对类的内聚性进行归一化的度量，LCOM*的计算需要分析每个类中方法与实例变量之间的关系，然后通过归一化公式进行计算。具体地，对于每个实例变量，需要统计该类中对其使用的方法数，并由此计算，该类中所有实例变量被访问的平均值，公式如下：

$$\mathrm{LCOM}^* = \frac{\left(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)\right) - m}{1-m}$$

式中：m 为方法数；a 为所含的实例变量数；$\mu(A_j)$ 为访问每个实例变量的方法数。从上式可以看出，LCOM*的取值介于 0 到 1 之间。如果每个实例变量在任何方法中都使用过，这时的 LCOM*为 0，然后随着每个变量在各个方法中使用程度的下降，对应的 LCOM*值会缓慢增加，直到一种极端情况，即每个变量只在一个不同的方法中被使用，此时 LCOM*为 1。所以，当 LCOM*为 0 时，该类的内聚性最好，当 LCOM*为 1 时，该类的内聚性最差。类的内聚性示意如图 11.12 所示。当然，若该类只有一个唯一的实例变量，则不需要考虑它的 LCOM*。

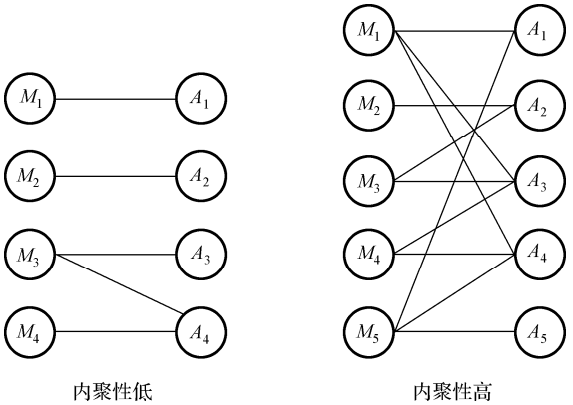


图 11.12　类的内聚性示意（*M* 表示方法，*A* 表示实例变量）

程序代码 11.3 对应的 $LCOM^{*}=\frac{\frac{1}{3}(3+2+1)-3}{1-3}=0.5$。由于 get 和 set 方法一般只对一个变量进行访问，为降低它们对 $LCOM^{*}$的影响，计算时可不考虑它们。另外，度量最好借助工具自动计算，工具应具有配置功能，可以对指标的软边界和硬边界进行指定，并且可以通过灵活配置，允许某些指标的超出。

代码 11.3　用以计算 $LCOM^{*}$的示例程序

```
public class LCOMExample{
    private int a;
    private int b;
    private int c;
    public void do1(int x){
        a=a+x;
    }
    public void do2(int x){
        a=a+x;
        b=b-x;
    }
    public void do3(int x){
        a=a+x;
        b=b-x;
        c=c+x;
    }
}
```


11.4　等价类测试

测试设计的主要工作之一就是寻找测试用例，通过对输入的描述、执行条件(被测对象状态)及预期结果对测试的具体内容进行说明，其目标就是使用这些测试用例高效地发现程序缺陷。本节介绍测试用例的设计方法，也是功能测试的基本方法，即等价类测试。由于等价类测试方法主要关注的是被测对象向外界提供的功能，测试用例的设计并不依赖程序内部结构，因此是一种黑盒测试的方法。

11.4.1　等价类方法

如何选取测试用例可以说是一门艺术，不同的测试人员会有不同的考虑。从单元层或者类中单一方法到活动图中可能的路径，等价类的思想都可以应用，以进行测试用例的设计。

等价类是离散数学中的一个概念，其基本思想是将一个集合按照一定的标准划分为若干个子集合，其中每个元素的归属依赖于指定功能下具体的行为。例如，按照全体自然数被 3 除的余数情况，可以将自然数分为 3 个等价类，第一个是余数为 0 的自然数集合{3,6,9,…}，第二个是余数为 1 的自然数集合{1,4,7,…}，第三个是余数为 2 的集合{2,5,8,…}。等价类有两

个重要的特性，即完备性和不相交性，也是测试设计的数学基础。完备性是指所有等价类的合集构成了输入域或值域的全集，这保证了生成的测试用例都具有针对性，没有遗漏；不相交性是指各个等价类子集之间没有交集，这保证了测试用例没有冗余。

等价类具有一个合理的假设，即每个等价类中的一个典型值在测试中的作用与这一类中所有其他值的作用相同，因此可以从每个等价类中只取一组数据作为测试数据，而且这样选取的测试数据也最有代表性，最可能发现程序中的错误。

将等价类的思想应用到类中方法或者功能上，主要的目的是从需求描述中尽可能准确地推导出程序处理的不同情况。以一个方法为例进行说明，假设该方法的输入为 0～ 100 的整数并产生某种可能的输出。首先对输入区间进行准确的界定，这里假设为开区间(0,100)，即 x 的合理取值范围为 $0<x<100$，其他整数为无效输入。无效的输入应该根据需求说明中的描述进行例外处理，可以把所有无效的输入视为无效等价类，对于数值类型的输入，可进一步将其分解为两个无效等价类，一个是“$x<1$”的区间，另一个是“$x>99$”的区间。在得到初步的等价类划分后，还可以根据具体需求对等价类进行进一步细化，然后针对每个等价类区间选取其中的一个典型值作为测试用例，用以代表整个等价类。上例中由于存在一个有效等价类和两个无效等价类，因此可以产生的测试用例输入是–3、8 和 102。

等价类的划分对于非数值类型的集合稍微复杂一些，对于枚举类型，假设合理的取值包括 red、yellow 和 blue，则可以简单地将这些取值分别对应一个等价类。若不允许其他值作为输入数据，则不存在它的无效等价类。

例如，考查一个类的构造方法，它用来创建某学生对象，具有名字、出生年份和专业三个参数。要求名字不能为空，出生年份介于 1900 年到 2000 年之间，专业是枚举类型，包括“贸易(TRADE)、计算机科学(CS)、数学(MATH)”。对于输入数据可产生如下的等价类：

(E1)名字非空(有效)
(E2)名字为空(无效)
(E3)出生年份小于 1900(无效)
(E4)出生年份大于等于 1900 并且小于等于 2000(有效)
(E5)出生年份大于 2000(无效)
(E6)专业为贸易(有效)
(E7)专业为计算机科学(有效)
(E8)专业为数学(有效)

这些等价类都要参与测试用例的设计。由于该方法具有三个输入参数，在对该方法进行调用时，要同时指定这些参数，因此总的测试用例的数量与这些等价类的组合相关。一种传统的组合规则是，对于有效等价类，要尽可能采用少的测试用例进行覆盖，如对于 E1、E4 和 E6 这三个有效等价类，可使用一个测试用例同时覆盖。对于无效等价类，则要慎重一些，其覆盖的规则是每个无效等价类必须与其他有效等价类组合测试，并且每个组合中只能含有一个无效等价类。这样可以保证触发该无效值对应的专门处理过程。

按照以上原则，可以产生如表 11.2 所示的测试用例。那些使用括号括起来的等价类表示已经被其他的测试用例覆盖过。

表 11.2　基于等价类的测试用例

测试用例	1	2	3	4	5	6
覆盖的等价类	E1 E4 E6	(E1) (E4) E7	(E1) (E4) E8	E2 (E4) (E6)	(E1) E3 (E6)	(E1) E5 (E6)
名字	“杨楠”	“田亮”	“王东”	“”	“杨楠”	“杨楠”
出生年份	1987	1989	1985	1988	1892	2006
专业	TRADE	CS	MATH	TRADE	TRADE	TRADE
预期结果	ok	ok	ok	fail	fail	fail

11.4.2　等价类与边界

缺陷经常会在边界处发生，等价类的边界应该得到格外的关注。例如，出生年份为 1988 的输入一定会产生错误，而对于 2000 这个边界，开发者则可能会有些模棱两可，做出自己认为有效或无效的假设。为此，在等价类的基础上，可以继续应用边界值分析的方法，具体做法是：对于每个等价类，继续测试其所含的边界，使用测试用例覆盖每个边界点及边界附近的情况。对于数值类型的等价类，其上下边界很容易确定，如 E3 的上边界 1899 和 E5 的下边界 2001，E4 的上边界 2000 和下边界 1900。

表 11.3　加入边界值分析的等价类测试用例

测试用例	1	2	3	4	5	6	7	8	9	10
覆盖的等价类	E1 E4D E6	(E1) E4D+ (E6)	(E1) E4U E7	(E1) E4U– (E7)	(E1) E4 E8	(E1) E3U (E6)	(E1) E3 (E6)	(E1) E5D (E6)	(E1) E5 (E6)	E2 (E4) (E6)
名字	“杨楠”	“杨楠”	“田亮”	“田亮”	“王东”	“杨楠”	“杨楠”	“杨楠”	“杨楠”	“”
出生年份	1900	1901	2000	1999	1985	1899	1892	2001	2006	1988
专业	TRADE	TRADE	CS	CS	MATH	TRADE	TRADE	TRADE	TRADE	TRADE
预期结果	ok	ok	ok	ok	ok	fail	fail	fail	fail	fail

另外还需要注意计算机中数值的表达方式，不同的数值类型，其能够表达的最小值和最大值也不同。若在程序中存在对这些值的特殊处理，则在边界值分析时需要仔细分析它们的边界值。例如，在浮点类型的数据处理中，需要注意 0 值和最小的正负数之间总是存在一个间隔，因此，在这种情况下需要对这些非零的最小数值也进行测试。

表 11.3 中给出了加入边界值分析的等价类测试用例，字母 D 表示下边界，字母 U 表示上边界，字母后的加号或减号表示边界附近。从表中可以看出，测试用例的数量明显增加了，这说明测试的精细程度也在提升。但如果仔细分析会发现，表 11.2 和表 11.3 中无效等价类的测试用例数量都占了一半，说明程序健壮性是传统的等价类方法的测试重点。

接下来，每个测试用例可以在一个独立的单元测试过程中实现，如用例 1 和用例 10 可以使用代码 11.4 实现。

代码 11.4　测试用例的实现

```
public void test1(){
    try{
```

```
            new Enrollment("杨楠", 1900, Major.TRADE);
        } catch(EnrollmentException e){
            fail("False Exception");
        }
    }
    public void test4(){
        try{
            new Enrollment("", 1988, Major.TRADE);
            fail ("False Exception");
        } catch(EnrollmentException e){
        }
    }
```

11.4.3 等价类组合

除了传统等价类方法的覆盖规则，还有其他的等价类组合方式。例如，将三个变量对应的等价类分别进行组合，就会产生 2×3×3=18 个测试用例，这将涵盖所有可能的输入情况，这种组合方式称为强等价类方法。这种产生测试用例的设计方法也比较容易实现，但前提条件是各个输入参数彼此独立。随着等价类数目的增加产生的测试用例数量也会急剧地增加，因此在实际测试设计中该方法具有一定的局限性。另外一种组合方式就是不采用组合覆盖的方式，而是要求每个等价类只覆盖一次即可，即弱等价类方法。弱等价类方法同样要求参数的独立性，它可以有效地降低测试用例的数量，但可能会有测试精度不高的问题。

如果输入变量彼此间存在相互依赖的关系，如计算给定日期的下一天的问题中，除了每个输入的合理区间，输入参数(年、月、日三个整数)彼此间的相互依赖关系也非常明显，如闰年、闰月的情况。如果在用例设计中忽略了这些输入参数间的依赖关系，那么就可能组合出过多的无效用例，从而使设计失去业务的针对性。

考虑变量之间存在依赖关系的实际被测代码 11.5 中的 Max 方法，其预期输出为三个变量中的最大值。若使用简单的等价类，先不考虑变量之间的关系，则每个输入参数都可以任意取值，因此三个输入变量分别对应一个 $(-\infty,+\infty)$ 的有效等价类。所以只需要一个测试用例即可覆盖所有的等价类，如{x=7, y=5, z=4)}。而且这时程序的输出值为 7，与预期一致。但是如果考虑到这三个变量之间及其与输出结果的关系，其实可以存在三种不同的情况，即最大值分别在首位、中位或尾位，相应的测试类也可以写成代码 11.5 中测试用例的形式，这是一种使用 JUnit 框架对用例进行实现的形式，后面还会进行介绍。

代码 11.5　等价类的覆盖问题

```
//Maxi.java
public class Maxi{
    public static int max(int x, int y, int z){
        int max=0;
        if(x>z) max=x;
        if(y>x) max=y;
        if(z>y) max=z;
        return max;
```

```
    }
}
//MaxiTest.java
import junit.framework.TestCase;
public class MaxiTest extends TestCase{
    public void testFirstMax(){
        assertTrue( "Maximum at 1st position", 7==Maxi.max(7,5,4));
    }
    public void testSecondMax(){
        assertTrue( "Maximum at 2nd position", 7==Maxi.max(5,7,4));
    }
    public void testThirdMax(){
        assertTrue( "Maximum at 3rd position", 7==Maxi.max(4,5,7));
    }
    public static void main(String[] args){
        junit.swingui.TestRunner.run(MaxiTest.class);
    }
}
```


这些测试用例目前都能成功运行，可以在此基础上针对变量间的关系对等价类进行细化，按照位置不同进行组合，也可以加入三个输入变量相等的情况，把这些都考虑完全后，组合后的等价类如下：

```
x>y=z    y=z>x    y>x=z    x=z>y    z>y=x
y=x>z    z>y>x    z>x>y    y>z>x    y>x>z
x>z>y    x>y>z    x=y=z
```

为每种情况设计其测试用例并执行后，其中的几个用例就会导致错误的出现，如下面的用例就会产生一个错误的输出结果 5。

```
public void testXZY(){
    assertTrue("x>z>y", 7==Maxi.max(7,4,5));
}
```

这个简单的例子说明了全面系统地设计出能够揭示所有问题的测试用例是一项复杂且耗时的工作。测试只是一种确认程序中在特定的形式下不存在缺陷的手段，但不能用来作为程序正确性的保证，这种保证需要更严谨和形式化的方法。

11.4.4 面向对象中的等价类

最后，我们考虑将等价类的方法应用到面向对象领域中。对象作为一个有机的整体，具有静态的属性和动态的方法，我们已经对类的方法(包括构造方法)应用了等价类方法进行测试，但类作为一个整体，在使用等价类方法进行测试时，要将状态作为一种输入参数考虑，类的状态对于其行为也有着重要的影响，因此首先要确定类的状态集合，然后据此进行等价类的划分。

类的状态是由其静态属性确定的，即实例变量。实例变量的数量决定了类的状态数量及

其等价类的复杂程度。为了简化类的状态和测试说明，下面的例子中只使用了一个实例变量。

在一个预订系统中，每个客户的信用等级通过一个专门的信用类 Credit 进行管理，该信用等级与客户的支付行为相关。信用类 Credit 提供了一个方法，对超过一定金额的订单按照客户信用情况进行资金流动性的检查。对于信用为 APPROVED 的客户，直接返回确定的结果，而对于信用为 CRITICAL 的客户，若没有进一步的验证，则返回否定结果，其余的客户需要对 500 元以上的订单进行进一步的验证，具体见代码 11.6 中的 Credit 类实现。

若使用单纯的等价类方法，则主要关注方法的输入参数 value，在这里为 500，因此可以产生两个有效等价类区间，小于 500 或大于等于 500。但这样的划分并没有考虑到用户的信用等级对业务逻辑带来的影响，因此，在对该类进行测试时，要同时考虑三个状态对应的三个有效等价类。这样，我们得到两个独立变量的等价类划分，即订单的金额及客户的等级，然后配合边界值分析方法，确定订单金额的两个等价类边界值分别为 499 和 500。若采用简单的组合覆盖(弱等价类)方法，则可以产生三个测试用例，分别是对象状态的三个等价类配合两个订单金额的边界值。

另一方面，我们注意到这个例子中的两个输入参数的组合对结果的作用，尤其是对于普通客户等级配合 499 和 500 的业务含义。考虑到本例中等价类的数目相对较少，而且在实现中对于另外两个客户等级在 500 边界处的订单处理逻辑也存在出错的可能性，因此可以采用完全组合的方法(强等价类)进行全覆盖，这样就会产生 6 个测试用例，对应的实现如代码 11.6 中的测试用例部分所示。

代码 11.6　Credit 类的等价类测试

```
//Paystatus.java
public enum Paystatus{
    STANDARD, APPROVED, CRITICAL;
}
//Credit.java
public class Credit{
    private Paystatus status;
    public void setStatus(Paystatus status){
        this.status=status;
    }
    public boolean checkPurchaseTotal(int value){
        switch(status){
            case APPROVED:
                return true;
            case STANDARD:
                return value<500;
        }
        return false;
    }
}
//CreditTest.java
import junit.framework.TestCase;
public class CreditTest extends TestCase{
```

```
    private Credit credit;
    protected void setUp()throws Exception{
        super.setUp();
        credit= new Credit();
    }
    public void testApproved1(){
        credit.setStatus(Paystatus.APPROVED);
        assertTrue(credit.checkPurchaseTotal(499));
    }
    public void testApproved2(){
        credit.setStatus(Paystatus.APPROVED);
        assertTrue(credit.checkPurchaseTotal(500));
    }
    public void testCritical1(){
        credit.setStatus(Paystatus.CRITICAL);
        assertTrue(!credit.checkPurchaseTotal(499));
    }
    public void testCritical2(){
        credit.setStatus(Paystatus.CRITICAL);
        assertTrue(!credit.checkPurchaseTotal(500));
    }
    public void testStandard1(){
        credit.setStatus(Paystatus.STANDARD);
        assertTrue(credit.checkPurchaseTotal(499));
    }
    public void testStandard2(){
        credit.setStatus(Paystatus.STANDARD);
        assertTrue(!credit.checkPurchaseTotal(500));
    }
}
```


等价类分析的方法也可以应用在输出参数上，然后反向考虑哪些输入能够产生对应的结果，也称输出等价类，其主要关注输出结果是由于哪些不同的输入引起的，如引发某个Exception 的输入情况。

11.5 基于控制流的测试

对于开发者来说，通常需要了解程序内部的结构，如选择某种特定的循环和分支结构来实现与业务逻辑之间对应的正确性，把基于这种思想的测试称为白盒测试。白盒测试的思想就是充分利用程序的结构信息设计测试用例，以实现对每个程序块(代码)的覆盖，并可方便地计算覆盖程度。代码覆盖率及相关指标通常在一些黑盒方法中是无法保证和度量的(即使等价类测试用例中已经蕴含了程序逻辑中的分支情况)。本节介绍几种基于控制流的测试方法及其覆盖指标的计算。

测试覆盖分析是指提供一组测试用例，尽可能使覆盖率指标越大越好，更准确地说，覆

盖率越接近 1 越好。显然，这里的覆盖率不是用来直接衡量产品本身的质量，而是一种用来评价测试质量的指标。覆盖率指标有很多不同的计算标准，其中较基础的有语句覆盖(Statement Coverage)、分支覆盖(Branch Coverage)、条件覆盖(Condition Coverage)、多条件组合覆盖(Multiple Condition Coverage)及路径覆盖(Path Coverage)等，接下来分别对它们进行介绍。

1. 语句覆盖

语句覆盖表示在程序控制流图中测试经过的节点数与所有节点数的比例，定义如下：

$$\text{语句覆盖}=\frac{\text{控制流图中测试经过的节点数}}{\text{所有节点数}}$$

在图 11.13 中，当测试输入为–1 时，其语句覆盖指标为 5/6；当测试输入为 0 时，其语句覆盖指标为 2/3，当测试输入为 1 时，对应的语句覆盖指标为 1(完全覆盖)，测试用例{–1,2}的语句覆盖指标也一样为 1(完全覆盖)。为了简化对问题的说明，这里省略了每个用例的预期输出。

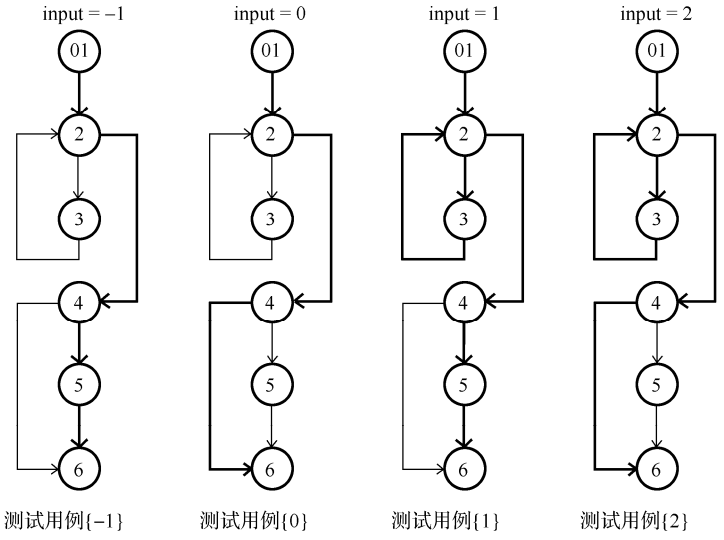


图 11.13　语句覆盖示例

2. 分支覆盖

语句覆盖是一种很粗略的度量，因为它主要关注的是控制流图中的节点，而不是节点之间的边。因此测试用例{1}虽然是完全覆盖，但并没有涵盖所有可能的边，如节点 4 到节点 6 的边，就没有被包含进来。正是由于这个原因，引入了分支覆盖，其目标是尽可能覆盖控制流图中所有的边，定义如下：

$$\text{分支覆盖指标}=\frac{\text{控制流图中测试经过的边数}}{\text{所有的边数}}$$

表 11.4 中给出了几组测试用例对应的分支覆盖指标。对于覆盖率指标，一般需要通过某种软件工具自动计算，尤其是对于逻辑较复杂的大型系统来说，通过手工进行覆盖率指标的度量在实践中是不现实的，因为这需要对源代码进行大量的、耗费成本的标注。在实际的开发中，分支覆盖的意义在于它要求对所有程序片段间的各种可能的连接至少执行一次，因此，满足分支覆盖要求一定会满足语句覆盖要求。

表 11.4　分支覆盖示例

测试用例	{−1}	{0}	{1}	{2}	{−1,0}	{0,1}	{1,2}	{−1,1}
分支覆盖指标	4/7	3/7	6/7	5/7	5/7	7/7	7/7	6/7

3. 条件覆盖

即使是分支覆盖，也无法保证理论上所有可能的程序逻辑都会被测试到。例如，观察 if 语句 if(a||b)，若 b 取 false，a 一次取 true，一次取 false，就可以达到完全的分支覆盖，因为两个测试用例将会覆盖 if 引出的两个分支。但是，两个测试用例都没有对 b 的取值重点关注，因为 b 没有取到 true。这就引出了另外一种覆盖标准：要求每个原子谓词的真假两种取值都要取到，即条件覆盖。条件覆盖不是根据程序的运行情况，而是根据出现的布尔条件进行测试用例的设计，条件覆盖与分支覆盖没有直接的关系，定义如下：

$$\text{条件覆盖指标}=\frac{\text{取值为真的原子谓词}+\text{取值为假的原子谓词}}{2\times\text{所有的原子谓词数}}$$

这里的原子谓词是指在程序中不能继续再拆分的布尔表达式，如上例中条件 if (input<0||input%2==1)中含有两个原子谓词，再加上 i>0 的原子谓词，共有三个原子谓词。

在一些高级编程语言中，如 Java、C#、C++、C 等，存在一个特别的问题，它们在进行条件判断时多采用短路(Short-Circuit)计算的方式，其含义为，为 a 取值为 true，则可以不进行 b 的评估，因为此时无论 b 取值如何，整个条件的判定取值都为 true。这种对条件的评估方式能够节省计算时间并且在逻辑上也是正确的，因为对整个条件的判断在进行第一步的评估后就可以得出结论了。但问题是，由于构造上的原因，b 可能存在某些缺陷，并由于无法测试到它们，从而产生某些技术上的副作用。例如，条件 if（x<4||x/0==2），当 x 满足第一个原子谓词时，由于短路评估方式的作用，将不会继续评估第二个原子谓词，而直接产生为真的结论，运行过程也不会有任何错误提示。在 Java 中，可以通过在条件判断时使用“|”来代替“||”，使用“&”来代替“&&”阻止这种短路计算方式。

表 11.5 给出了不同的测试用例集合及其对应的条件覆盖情况，t 表示 true，f 表示 false，短横线表示由于短路评估方式不需进行计算。这里需要再次强调的是，条件覆盖与分支覆盖没有任何特别的联系，这表示分支的完全覆盖不能保证条件的完全覆盖，反之也成立，即完全的条件覆盖也不能保证分支的完全覆盖。例如，条件 if(a|b)中，{a=true, b=false}或者{a=false, b=true}两个用例能够满足条件的完全覆盖，但两种情况下整个判定的取值都是 true，也就是说，该 if 语句的另外一个分支并没有被覆盖，从而不满足分支覆盖的标准。同样，如果从分支覆盖的要求出发，用例{a=true, b=false}及{a=false, b=false}能够覆盖所有的节点和边，满足分支覆盖的要求，但却不满足条件覆盖的要求。

表 11.5　条件覆盖示例

测试用例	{−1}	{0}	{1}	{2}	{−1,0}	{0,1}	{1,2}	{−1,1}	{−1,1,2}
i>0	f	f	f,t	f,t	f	f,t	f,t	f,t	f,t
input<0	t	f	f	f	t,f	f	f	t,f	t,f
input%2==1	–	f	t	f	f	f,t	t,f	t	t,f
条件覆盖指标	2/6	3/6	4/6	4/6	4/6	5/6	5/6	5/6	6/6

在分支覆盖和条件覆盖的基础上，还衍生出一种条件/分支覆盖标准，它是两种覆盖的混合，要求同时满足两种覆盖的要求。

4. 多条件组合覆盖

最后，所有的覆盖要求在多条件组合覆盖标准中得到了综合，它要求所有在条件中出现的原子谓词的组合都要覆盖到。例如，条件(a||b)&&(c||d)，需要覆盖{a, b, c, d, (a||b), (c||d), (a||b)&&(c||d)}这 7 种不同的谓词取值。多条件组合覆盖的定义如下，即满足要求的最小条件组合覆盖为：

$$\text{多条件组合覆盖指标}=\frac{\text{取值为真的原子谓词数}+\text{取值为假的原子谓词数}}{2\times\text{所有的原子谓词数}}$$

需要注意的是，条件的整体(a||b)&&(c||d)也要作为一个谓词参与计算。表 11.6 给出了上例中多条件的覆盖的用例集合和对应指标。从定义中可以得出结论，在非短路计算的情况下，满足多条件覆盖的要求同时会满足条件覆盖、分支覆盖和语句覆盖。

表 11.6　多条件覆盖示例

测试用例	{-1}	{0}	{1}	{2}	{-1,0}	{0,1}	{1,2}	{-1,1}	{-1,1,2}
i>0	f	f	f,t	f,t	f	f,t	f,t	f,t	f,t
input<0	t	f	f	f	t,f	f	f	t,f	t,f
input%2==1	—	f	t	f	f	f,t	t,f	t	t,f
input<0\|\|input%2==1	t	f	t	f	t,f	f,t	t,f	t	t,f
多条件组合覆盖指标	3/8	4/8	5/8	5/8	6/8	7/8	7/8	6/8	8/8

5. 路径覆盖

路径覆盖(Path Coverage)是度量一个方法中所有可能路径的覆盖情况。路径是指从方法的入口开始到出口结束，由控制流图中若干条边构成的一条唯一的执行路线，也可以看成由若干可能的逻辑条件的组合。

由于循环结构会使得路径的数量剧烈地增长，因此路径覆盖只考虑有限循环次数的情况，例如，只考虑循环 1 次和 k 次(k>1)的情况。基本路径测试是路径覆盖的典型方法，它是在控制流图的基础上，通过分析控制结构的环形复杂度，导出基本可执行的路径集合，从而设计测试用例的方法，主要步骤如下。

(1)绘制程序的控制流图。

(2)计算 McCabe 环形复杂度。从程序的环形复杂性可导出程序基本路径集合中的独立路径条数，独立路径要求在路径中至少含有一条未曾使用过的边。

(3)导出测试用例。为每一条基本路径设计测试用例的数据输入和预期结果，并确保覆盖到基本路径集中的每一条路径，参照环形复杂度规定的上限路径条数。

控制流图和环形复杂度的计算在这里不再赘述，具体见章节 11.3.1。作为示例，给出控制流图 11.11 对应的测试用例。通过计算已经得出该图的 $V(G)$=4，也就是基本路径条数的上限是 4 条。

示例中的所有路径如表 11.7 所示，其中循环“2→3→2”可能会执行多次，这里只显示一次。从该表中可以清晰地看出，用例 1、用例 2、用例 5 及用例 6 是满足基本路径覆盖要求的测试用例；用例 3 和用例 4 对应的都是不可达路径，因此无法找到实际的输入数据。

表 11.7　示例中的所有路径及其测试用例

用　例	路　径	输　入	预期输出
1	01→2→4a→5→6	{-1}	0
2	01→2→4a→4b→6	{0}	0
3	01→2→4a→4b→5→6	—	—
4	01→2→3→2→4a→5→6	—	—
5	01→2→3→2→4a→4b→6	{2}	4
6	01→2→3→2→4a→4b→5→6	{1}	0

这里只是介绍了常见的几种覆盖方法，在相关的文献中还有更多的覆盖方法。由于对于覆盖情况的度量较为复杂，目前只有少数工具支持高级覆盖指标的自动计算。覆盖指标要求提升一点，如从 96%提高到 97%，在测试用例设计上往往需要付出较高的成本。在一些关键软件系统(如金融、军工等领域)的开发和测试中，这些覆盖指标都有着广泛的应用。

为快速达到覆盖指标的要求，可以借鉴一些相关的策略。一是可以把精力全部投入到完全覆盖测试的设计上，但这成本比较高；另外还可以考虑先利用其他方法进行测试，如等价类方法，并计算其覆盖情况，如果未达到要求，再使用白盒的覆盖测试进行补充。同时，可以参照以下内容进行分析：

(1)等价类构造的数量不足，太少的输入等价类可能会导致某些逻辑分支无法被测试到。

(2)存在问题的编程风格，如方法过长或者过多的选择分支，会使方法逻辑趋于复杂，从而降低可测试性。

(3)多余的开发内容，如开发者凭借自己的假设开发出的内容在运行时从来没有到达过。这里需要慎重地分析，这些内容是否真的没有用处，需要考虑未来的扩展，判断其是否需要继续在代码中保留。

覆盖测试的方法适用于所有能够通过控制流图等进行描述的内容，这意味着对于活动图或者状态图等描述的业务，测试同样也可以通过覆盖方法进行。

11.6　断言机制

当前很多高级编程语言都提供用来进行断言的指令，例如在 Java、C#和 C++中，一般形式为 assert <Boolean condition>，这就是可以在不同的场景中使用的断言机制。需要注意的是，虽然它们提供了一种对异常进行检查的能力，但只能在编码阶段使用，并且不能替代常规的异常处理。

断言主要的作用是在某些关键的位置确认某种假设在程序中是正确的。因为断言要占用存储空间和运行时间，因此很多编程语言都提供对断言的一些编译开关，使其在编译时可以处于有效或无效状态。断言指令可以处于方法的任何位置，如代码 11.7 所示的是一个处于方法开始处的断言。

代码 11.7　方法开始处的断言

```
public void analyse(int a, int b){
    assert a>b && exemplar>5;
    //...
}
```

这个断言表示，该方法要求其参数满足相应的条件并且该对象需要处于某种状态，即实例变量 exemplar 的值需要大于 5。不要将这种假设与异常相混淆，断言的加入使方法必须要满足既定的条件。若对于输入的内容不能确定是否满足既定的要求，例如在防御性程序设计(Defensive Programming)[①]中要求方法总是可用，则必须要使用异常处理，而不能使用断言。Java 中的断言指令在代码 11.7 的基础上进行了扩展：

```
assert <boolean condition>: <any object>
```

表示如果断言不能得到满足，给定的对象<any object>将被输出。通常，该对象为一个 String 类型或者是实现了 toString()方法的对象。

位于方法尾部的断言可用来检验该方法的计算是否是期望的结果，或者说，计算出的结果是否具有期望的某些属性。代码 11.8 使用一个乘法的过程进行说明，其中的两个参数必须为正整数。

代码 11.8　程序尾部的断言

```
public int multiplication(int a, int b){
    int op1=a;
    int result = b;
    int rest = 0;
    assert(a >= 1 && b >= 1): "not positive parameter";
    while (a > 1){
        result = result*2;
        if( a%2 == 1 )
            rest = rest + b;
        a = a/2;
    }
    result=result + rest + b;
    assert result == op1*b: "false computation";
    return result;
}
public static void main(String[] args){
    Computer r = new Computer();
    System.out.println("7*9="+r.multiplication(7,9));
    System.out.println("6*8="+r.multiplication(6,8));
    System.out.println("6*0="+r.multiplication(6,0));
}
```

上面的方法 multiplication 中加入了两个断言，假设该方法位于 Computer 类中，对该方法的测试需要传递合适的参数并在 main()函数中进行调用。对应的输出如下：

```
7*9=63
6*8=48
Exception in thread "main" java.lang.
AssertionError: not positive parameter
at Computer.multiplication(Computer.java:8)
at Computer.main(Computer.java:24)
```

① 防御性程序设计是一种细致、谨慎的编程方法。通过对未来问题的预见设置环境，进行捕获和处理。

如果断言的编译开关被关闭，断言相当于注释，它们也不再占用任何附加的资源。但这也不是绝对的，因为那些只在断言中使用的信息往往还要占用一定的资源，如上面乘法功能中的变量 op1，它只在最后一个断言中被使用。

这里有一个使用断言的小技巧：assert false 断言好像没有什么用处，因为其永远都不会通过，它的作用在于放置于认为不应该到达的地方，以确保此处不可达。如代码 11.9 所示，确保了只有 65 岁以下的人可以参与到折扣计算中。

代码 11.9　assert false 的用处

```
public int discount(int age){
    int result=0;
    if(age>=0 && age<18)
        result=10;
    else if(age>=18 && age<65)
        result=5+(age/2);
    else
        assert false: "false age";
    return result;
}
```


另外，断言的使用，要确保不会带来任何副作用。也就是说，不会改变实际类的状态。例如，assert iter.next()!=null 中对迭代器 iter 的断言是不合适的，因为该断言被检测后会导致迭代器状态的改变，使其指向了下一个对象。

11.7　测试框架

软件测试中一个非常重要的任务就是设计能够尽可能发现问题的测试用例，而测试的执行则是对测试用例的驱动，根据测试用例执行的结果判断是否存在缺陷。另外，当测试用例较多的时候，需要使用某种手段对测试用例进行管理，以便更加有效地组织和实施测试。测试框架提供了一种对测试进行自动化管理的环境。

11.7.1　测试环境

测试的一般方法就是针对待测系统(System Under Test，SUT)进行分析和尝试，并确定它是否按照期望的方式运行。测试用例来自已经建立的规格说明书，规格说明书必须能够清晰地描述期望的行为。同时，规格说明出又源自用户需求，因此通过测试用例的执行可以逐步完成用户需求的测试。在这个过程中，活动图是非常有帮助的，因为它代表了业务的执行路径。

文本形式的需求需要尽可能精确，同样各个需求之间的联系也要准确建立，这对于测试用例的设计是非常重要的，因为每个测试用例在执行前都要确定出它的前提条件或应用环境。例如，对某个类中的简单方法进行单独测试，可能搞清楚方法的每个输入参数的说明即可，但若该方法引用了其他实例变量或其他对象，则在测试前需要确定好相关变量的正确取值。测试设计中把这样的工作称为测试配置，它描述了测试时其他相关部分应具有的状态。测试配置主要关注的内容包括：

(1) 目标系统运行的硬件，从硬件上的操作系统到硬件层本身与测试环境都是相关的。

(2) 与目标系统在运行环境中同时执行的软件，这个与系统的性能分析有重要联系。

(3) 目标系统将会与其他软件的哪些版本一起工作，这些具体的版本应在测试环境中准备好。

测试环境的搭建应该与未来用户的使用环境情况越相近越好，因此，具有一个综合的测试实验室，能够实现各种用户环境的配置并快速实施测试是非常有意义的，但在现实中，需要对各种约束做出实际的妥协以模拟用户环境，尤其是对网络环境中集群式环境的模拟。近年来，出现了虚拟机的概念，使人们不再需要大量的实体计算机，并且可以省去各种软件安装的麻烦，可在操作系统级实现对虚拟机高效的管理，为测试环境的快速建立和灵活管理提供了便利。

同样，测试的过程也需要进行精确的描述。这种描述最简单的形式就是对过程调用的顺序进行安排。复杂一点的系统可能还要经常考虑时间条件，如被测系统在某个事件后，需要在指定的时间间隔内做出响应等约束。

最后，在测试用例中需要准确地给出期望的运行结果，以便能够比较容易地判断测试是否发现了某个缺陷。当然，有时对结果的描述也会花费较大的精力，这依赖于系统的复杂性。例如，数据库连接过程的测试，就需要描述出每个数据库操作的目标状态。

如果要实现对断言的测试，必须要将其置于一个测试环境中，如建立一个 main() 函数实现对目标函数的调用。但是，我们已经发现测试环境的创建将会随着测试需求及测试用例的增加变得更复杂和难以管理，因此，作为另外一种可选的方案，人们使用测试工具对测试用例进行管理。

11.7.2 JUnit

Kent Beck 提出并完成了第一个较为简单的测试框架，它允许使用各种程序设计语言创建测试，而不需要花费过多的精力在整个测试环境的搭建上。首先在 Smalltalk 平台上实现了一个 SUnit 框架，后来又在不同的语言平台上(如 Java、C#、C++等)进行了移植，它们称为单元测试框架。这里我们对 JUnit 的测试框架进行重点说明，主要基于基础的 3.8 版本。虽然目前 JUnit 的最新版本 JUnit 5，但 JUnit 4 目前应用得更为广泛，它是在 JUnit 3 的基础上主要增加了对 Java 5 及以上的支持，即加入了 Java 的 Annotation 机制。Kent Beck 也曾表示 JUnit 3.8 中的基本概念和使用形式将在其后版本中继续得到支持。

JUnit 的基本思想是对不同的测试用例创建与其对应的测试方法，测试用例的执行和评价由 JUnit 接管。在 JUnit 3.8 中，这些测试方法的基本形式为：

```
public void test<furtherNamePart>(){ ... }
```

每个测试用例具有自己不同的名字。JUnit 可以使用反射机制在测试环境中调用传递过来的方法，所有的 test() 方法都放在一个测试类中，这个类需要从 JUnit 提供的系统类 TestCase 继承而来。如果该类需要测试的内容为 X，其命名一般情况下采用 testX 的形式。

JUnit 负责执行所有的测试用例，JUnit 中还提供了一个附加的图形界面，其中显示出所有执行的测试数、通过的测试数及失败的测试数等，这个状态显示的界面现在已经在很多大型开发环境中得到了支持和无缝集成，并且当有某错误发生时，相应的错误信息也会显示出来，JUnit 也逐渐不再需要自己的界面了。

JUnit 的测试类也是一个普通的 Java 类，含有自己的实例变量和方法。对于测试来说，重要的是其测试方法的确定性，即对于相同的输入，一定会有相同的输出结果，测试执行的顺序不应对测试的结果造成影响。为了营造这样的环境，JUnit 测试类需要重写两个方法：

```
protected void setUp(){ ... }
protected void tearDown(){ ... }
```

setUp()方法在每个测试开始之前都要执行，其中可以对实例变量等环境相关的内容进行设置。同样，每个测试结束后会执行 tearDown()方法，它比较适合做一些清理工作。因为 Java 具有垃圾回收的能力，能够将一些不被引用的对象自动删除，因此，这个清理过程中可主要做一些诸如关闭文件或关闭数据库连接等工作。总结一下，测试方法 testX()的执行首先是 setUp()先执行，然后是 testX()本身执行，最后是 tearDown()执行。

在框架提供的测试方法中，也提供了对某些特征进行检验的特殊方式，这可以通过调用 TestCase 类的父类 Assertion 中提供的方法，其中较为重要的方法如下：

```
assertTrue(<Text>,<Boolean condition>);
```


这个方法对布尔条件进行检测，若为 false，则认为发现了一个错误，测试过程停止，并将文本部分连同错误出现的具体位置等信息由 JUnit 一同输出。若在文本信息中同时给出期望状态和实际状态，则会使该方法的使用更加具体。该方法还有一个简化版本：

```
assertTrue(<Boolean condition>);
```

其中省略了文本部分。除了 assertTrue()方法，还存在一些其他的检测方法，它们都可以视为 assertTrue()的特殊形式，具体包括：

(1) assertEquals(Object expected, Object actual)：检测两个对象是否一致，将会分别调用两个对象的 equals(Object)方法进行具体比较。

(2) assertEquals(int expected, int actual)：检测两个变量是否具有相同的值，同样具有对于简单类型 float、byte、char、short、long 和 boolean 的版本。

(3) assertEquals(double expected, double actual, double delta)：检测两个 double 类型值的差别是否在 delta 之内，同时也存在类型 float 的版本。

(4) assertSame(Object expected, Object actual)：检测两个对象是否完全一样。

(5) assertNull(Object object)：检测某个对象是否为空。

(6) assertNotNull(Object object)：检测某个对象是否不为空。

要进一步说明的是 assertEquals()方法的 delta 版本，因为对于浮点类型的精确比较是不合理的，该版本的断言提供了一种差值在给定范围内的比较方式，实用性更强。

除了 assert 类型的方法，还提供了 fail(<Text>)方法。它对某个不应到达的位置进行了标记，Text 会在错误发生时输出。该方法可以用来验证某个异常的处理是否正确，例如，测试方法 xy()，若希望方法会抛出某个期望的异常，则可以在测试中使用代码 11.10 中第 1 行的方法。若希望方法不会有异常抛出，可将测试中对于 fail 的调用置于异常处理中，如代码 11.10 中第 7 行的方法。

代码 11.10　fail 方法的用法

```
public void testXyThrowsException(){
    try{
```

```
            ob.xy();
            fail("ob did not throw any Exception" + ob);
        } catch (XYException e){}
    }
    public void testXyThrowsNoException(){
        try{
            ob.xy();
        } catch (XYException e){
            fail ("ob threw Exception" + ob);
        }
    }
```

下面介绍一个具体的示例。这里没有使用在每个测试函数开始前进行初始化的 setup()方法。该示例由一个折扣类 Discount 负责存储当前折扣情况，其中含有一个 discount 实例变量。另外，该类提供一个对某个客户“锁定”的能力，对锁定客户的折扣计算将会导致 DiscountException 异常的抛出。具体实现如代码 11.11 所示。

代码 11.11　Discount 类的实现


```
//异常类定义
public class DiscountException extends Exception{}
//Discount 类定义
public class Discount{
    private double discount;
    private boolean locked;
    public Discount(double discount, boolean locked){
        this.discount = discount;
        this.locked = locked;
    }
    public boolean isLocked(){
        return locked;
    }
    public void setLocked(boolean locked){
        this.locked = locked;
    }
    public double getDiscount(){
        return discount;
    }
    public void setDiscount(double discount){
        this.discount = discount;
    }
    double price(double originalprice) throws DiscountException{
        if(locked)
            throw new DiscountException();
        return originalprice*(1 - (discount/100));
    }
}
```

测试类 DiscountTest 可以写成代码 11.12 的形式。测试中，经常将一个大的测试过程分解为多个独立的测试用例顺序执行，这样的方式多是若干 assert 类型的方法先后出现，如果第一个 assert 方法产生错误，后续的方法将不会被继续执行。

代码 11.12　DiscountTest 类

```
package Discount;
import junit.framework.TestCase;
public class DiscountTest extends TestCase{
    private Discount good;
    private Discount bad;
    protected void setUp()throws Exception{
        super.setUp();
        good=new Discount(3.0,false);
        bad=new Discount(0.0,true);
    }
    public void testGetDiscount(){
        assertTrue(3.0==good.getDiscount());
        assertTrue(0.0==bad.getDiscount());
    }
    public void testSetDiscount(){
        good.setDiscount(17.0);
        assertTrue(17.0==good.getDiscount());
    }
    public void testIsLocked(){
        assertTrue(!good.isLocked());
        assertTrue(bad.isLocked());
    }
    public void testSetLocked(){
        good.setLocked(true);
        bad.setLocked(false);
        assertTrue(good.isLocked());
        assertTrue(!bad.isLocked());
    }
    public void testPriceSuccess(){
        try{
            double result=good.price(100.0);
            assertEquals(result,97.0,0.001);
        } catch (DiscountException e){
            fail("false DiscountException");
        }
    }
    public void testPriceWithException(){
        try{
            bad.price(100.0);
```

```
            fail("failed DiscountException");
        } catch (DiscountException e){
        }
    }
    public static void main(String[] args){
        junit.swingui.TestRunner.run(DiscountTest.class);
    }
}
```

在很多大型的项目开发中，需要对测试用例进行收集和管理，并在合适的时刻重复执行所有测试用例或者根据情况重复执行部分测试用例，这称为回归测试。JUnit 使用 TestSuite 的概念来支持对测试用例集合的管理。具体步骤为：首先创建类型为 TestSuite 的对象，然后通过该对象实现对指定测试用例的组织，如代码 11.13 所示。

代码 11.13　使用 TestSuite 类对测试用例进行管理

```
import junit.framework.Test;
import junit.framework.TestSuite;
public class AllTests{
    public static Test suite(){
        TestSuite suite=new TestSuite( "Clientsystem" );
        //从测试类加入测试用例集合
        suite.addTestSuite(DiscountTest.class);
        suite.addTestSuite(ClientTest.class);
        //也可以从其他的 TestSuite 加入测试用例集合
        suite.addTest(OtherAllTests.suite());
        return suite;
    }
    public static void main(String[] args){
        junit.swingui.TestRunner.run(AllTests.class);
    }
}
```

测试框架的主要作用是实现对测试用例的管理，一方面将独立运行的测试进行统一组织，在其执行前后加入配置能力；另一方面能够按照要求选择合适的测试分组管理。这种做法在 JUnit 4 中也同样支持，DiscountTest 类针对 JUnit 4 的移植代码如代码 11.14 所示。

代码 11.14　DiscountTest 类在 JUnit 4 中的实现

```
package Discount;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;
public class DiscountTestJUnit4{
    private Discount good;
```

```
private Discount bad;
@BeforeClass
public static void onetimeAtBeginning(){
    System.out.println("Open Database");
}
@AfterClass
public static void onetimeAtEnd(){
    System.out.println("Close Database");
}
@Before
public void start()throws Exception{
    //super.setUp(); 不再需要
    System.out.println("start called");
    good=new Discount(3.0,false);
    bad= new Discount(0.0,true);
}
@After
public void stop(){
    System.out.println("stop called");
}
@Test
public void getDiscount(){
    assertTrue(3.0==good.getDiscount());
    assertTrue(0.0==bad.getDiscount());
}
@Test
public void setDiscount(){
    good.setDiscount(17.0);
    assertTrue(17.0==good.getDiscount());
@Test
public void isLocked(){
    assertTrue(!good.isLocked());
    assertTrue(bad.isLocked());
}
@Test
public void setLocked(){
    good.setLocked(true);
    bad.setLocked(false);
    assertTrue(good.isLocked());
    assertTrue(!bad.isLocked());
}
@Test
public void priceSuccess(){
    try{
        double result=good.price(100.0);
        assertEquals(result,97.0,0.001);
    } catch (DiscountException e){
```

```
            fail("false DiscountException");
        }
    }
    @Test(expected = DiscountException.class)
    public void PriceWithException() throws Exception{
        bad.prise(100.0);
    }
}
```

以上代码与 JUnit 3 的区别首先是测试类不需要再从其他类中继承，使开发更灵活。另外，各测试方法的名字不再要求以“test”开头，而是通过 Java 5 中的 Annotation 注解机制使用了@Test 标识。另外，配置方法中不再要求名字必须是 setUp 或 tearDown，可以是任意的名字，使用@Before 和@After 标识。同时，新增了一种特殊的方法，其提供了对所有测试进行一次性初始化和结束清理的工作，并使用@BeforeClass 和@AfterClass 标识。

JUnit 4 中测试方法的写法也可以与 JUnit 3 中保持一致，不需要更多的改变，只是在其前面加上@Test 注解即可。对于期望异常的测试，可以在 Annotation 注解中将这个异常作为参数，如代码 11.14 中的 priceWithException()所示，重要的是，要在测试方法的头部加入异常的抛出动作 throws。

JUnit 4 中还存在@Ignore 标识，可以置于@Test 前面，表示在接下来的测试中不去运行该测试方法。对于方法的命名，如 setUp 和 tearDown，建议在 JUnit 4 中采用同样的方式，这样可以大大增加程序的可读性。

JUnit 是比较适合开发者对自己创建的单个类进行测试的工具。事实上，JUnit 存在一个强有力的竞争对手，TestNG 框架。两个框架的不同在于它们的核心设计。JUnit 一直是一个单元测试框架，也就是说，其构建目的是促进单个对象的测试，而且它确实能够有效地完成此类任务；而 TestNG 则用来解决更高级别的测试问题，它具有 JUnit 中没有的一些特性，如依赖性测试、参数化测试及多线程测试等特性。

11.8 可测试性

测试的一般策略是采用“自底向上(Bottom Up)”的构建方式，即从粒度较小的类方法开始测试，然后再对由这些简单方法构成的更为复杂的方法进行测试。如此进行下去，直到粒度较大的与其他类没有依赖关系的简单类测试完毕，再由这些简单类出发，测试那些与其相依赖的类。也就是说，新的测试总是在已经经过测试的类和方法的基础上进行构建，如要对一个依赖某个类的另外一个类创建测试，则无须在这个测试类中重新测试被依赖的类的 price()方法。

上述(集成)测试过程可能会由于过高的开销或遇到类之间较复杂的依赖关系而无法继续下去，这时需要在一个较高的层次上指定一个相应的测试策略。也就是说，要确定一种对多个类同时进行测试的方式。

在测试进行的过程中，尤其是在多人并行开发时经常会出现彼此依赖的情况。例如，某开发者开发的 A 类需要依赖其他开发者的开发成果 B 类，但 B 类还没有正式开发完毕。这时

为了避免等待而影响开发进度，可以通过构建 B 类的一个简化形式的 B’类来模拟 B 类，B’类称为 B 类的桩或模拟(Stub or Mock)，并且在构建 B’类时需要在满足 A 类需要的基础上尽可能地对其简化实现。

在模拟程序中，最简单的一种实现就是返回一个默认值，例如，一个返回 void 类型的方法，可以是一个完全为空的方法。如果需要返回一个实际的对象，提供一个空值 null 就足以使得相应的代码运行起来。如果返回类型为 int 或 boolean 等，那么需要给出一个相应类型的默认值，如 0 或 false。

桩的具体实现依赖于 A 类的实际需要,通常最简化的桩的实现不能完全满足 A 类的要求。例如，A 类需要代码 11.15 所示的一个返回布尔类型的桩。

代码 11.15　桩的实现示例

```
public boolean determine(int i){
    return false;
}
//或者
public boolean determine(int i){
    return i==40;
}
```


代码 11.15 中的第一个桩只能为 A 类一直提供返回值为 false 的情况，即 A 类只能测试到这一种情况。为此，可以对该桩采用第二个稍复杂的实现方式，它可以较灵活地控制结果的返回值。程序中的具体数值可根据 A 类的实际需要调整，以避免不断的修改。代码 11.16 对此进行了示意性描述。

代码 11.16　Transaction 类及其需要的桩

```
//Transaction.java
package TransactionMock;
public class Transaction{
    public static LogData logging;
    //...
    public synchronized void debit(int id, Account account, int amount)throws
                                        TransactionException{
        if(account.isLiquid(amount)){
            account.debit(amount);
            logging.write(id + " processed");
        } else{
            logging.write(id +" insolvent");
            throw new TransactionException("insolvent");
        }
    }
}
//TransactionException.java
package TransactionMock;
public class TransactionException extends Exception{
    public TransactionException(String s){
```

```
        super(s);
    }
}
//LogData.java
package TransactionMock;
public class LogData{ // Mock for Transaction
    public void write(String s){ }
}
//Account.java
package TransactionMock;
public class Account{ //Mock for Transaction
    public boolean isLiquid(int amount){
        return amount<1000;
    }
    public void debit(int amount){ }
}
```

代码中也包含了日志类 LogData 的实现，该类只需要一个没有返回值的方法，因此可以将其实现为最简单的形式。Account 类中需要 debit()和 isLiquid()两个方法。方法 debit()使用简单的实现方式即可，对于方法 isLiquid()，则需要思考一下，因为 Transaction 类需要其提供返回真、假两种情况，才能进行较充分的测试。因此，Account 类的桩实现可以写成代码 11.16 中的形式。现在可以编写 Transaction 类的测试代码了，相应的测试类如代码 11.17 所示。

代码 11.17　Transaction 类的测试实现

```
package TransactionMock;
import junit.framework.TestCase;
public class TransactionTest extends TestCase{
    private Account account;
    private Transaction transaction;
    protected void setUp()throws Exception{
        Transaction.logging=new LogData();
        transaction=new Transaction();
        account=new Account();
    }
    protected void tearDown()throws Exception{
        // logging close
    }
    public void testSuccessTransaction(){
        try{
            transaction.debit(42,account,100);
        } catch (TransactionException e){
            fail();
        }
    }
    public void testFailedTransaction(){
        try{
            transaction.debit(42,account,2000);
```

```
                fail();
            } catch (TransactionException e){}
        }
    }
```

通过上面示例可知，在测试程序构建的过程中，要经常通过桩来隔离被测程序。当项目较为复杂时，仅通过手工构建桩，将非常烦琐和低效。为此，一些专门用于模拟的自动化框架应运而生，使得构建被测程序更加简洁和快速，如 JMock、EasyMock 等框架。

另一方面，在被测程序的开发过程中，为了更容易地进行测试，就要为可测试性做好的准备。下面介绍几个可测试性构建的原则。

1. 设计简单的方法

带有很多规模较小的方法的类的测试性要好于那些带有较少方法但每个方法的规模较大的类。方法的规模越大，对其所有方面的测试就越困难。另外，方法的调用接口要尽量简单，这符合最少知识原则，它主张一个类只要知道它需要知道的那么多即可。下面的两个方法中，第二个方法的可测试性要更好。


```
getRent(car, customer)
getRent(car.getDays(), car.getPrice(), customer.getDiscount())
```

2. 避免私有方法

方法的可测试性是较差的，由于封装性，普通的 JUnit 类无法直接对它们进行访问。JUnit 4 之后的 protected 方法的情况将有所好转，因为一个测试类可以从任意类中继承，如可从测试的目标类继承。由于 private 的特点，在设计相关的类方法时，应考虑尽量使用它们来完成特别简单的任务。

3. 优先使用通用方法

静态代码无法应用多态的特性，也就意味着没有代码重用，无论是对被测程序还是测试程序。这就可能会导致应用程序与测试代码的冗余，也无法实现测试代码与程序代码的轻松替换。多态的另外一个好处是能够较容易地降低方法的复杂程度，避免使用较长的 switch 和 if 语句。

4. 组合优于继承

类之间通过关联进行组合的关系更易于测试，因为运行时，代码不能改变继承的层次结构，但可以组合不同的对象，使得对象易于从一种状态切换到另外一种状态，也就易于测试。

5. 避免隐藏的依赖关系与全局状态

对全局状态一定要谨慎，因为全局状态使测试的构建变得复杂，如果全局状态没能共享或者遗漏，都会导致一些意外的后果。隐藏的依赖关系也一样，这些都会使测试过程变得艰难。

代码 11.18 中的 Database 意味着一个全局状态。如果不首先实例化数据库，就无法进行付款的操作。从内部看，CreditCard 类直接或间接地依赖于 Database 类、OfflineQueue 类及 CreditCardProcess 类，这些依赖是隐藏的，使测试的构建困难重重。代码 11.19 通过依赖注入的方式增强了可测试性，测试的构建也就不那么困难了。

代码 11.18　实际测试中的全局状态

```
testCreditCardCharge(){
    Database.init();
    OfflineQueue.init();
    CreditCardProcessor.init();
    CreditCard c = new CreditCard("1234 5678 9012 3456", 5, 2018);
    c.charge(100);
}
```

代码 11.19　避免全局状态

```
testCreditCardCharge(){
    Database db = Database();
    OfflineQueue q = OfflineQueue(db);
    CreditCardProcessor ccp = new CreditCardProcessor(q);
    CreditCard c = new CreditCard("1234 5678 9012 3456", 5, 2018);
    c.charge(ccp, 100);
}
```


11.9　建设性质量保证

目前为止介绍的一些质量保证的方法，尤其是软件测试方法，是在软件开发后作用于被测软件的，用于发现缺陷的存在，它们通常称为分析式的质量保证方法(Analytical QA)。除此之外，借助于度量，还可以使用其他方法进行质量保证。一般来说，度量是针对已开发的产品的，但在开发前可以设置某些度量指标的期望值，如在开发前约定环形复杂度，最大不超过 6，那么所有的开发者就需要在程序设计中注意，使用较少的分支或者将较复杂的流程分解为简单的方法。

这些实际产品开发完成之前的质量活动称为建设性质量保证活动(Constructive QA)。这些活动可能针对的是具体的产品(如需求规格说明书或者程序代码)，也可能是整个开发过程，它们可以是如下内容：

(1) 原则(Principles)、技术(Technique)、方法(Methods)和工具(Tools)；

(2) 生命周期模型(Life-Cycle Models)；

(3) 文档(Documentation)；

(4) 需求(Requriements)：系统化的(Systematic)和正式的(Formal)表示；

(5) 开发环境(Development Environments)；

(5) 配置管理(Configuration Management)；

(6) 人件(Peopleware)。

例如，通过对过程模型的选择，能够确定每个工作环节与具体产品之间的关系，然后经过组织级的剪裁定制并制度化，最终形成过程指南。过程指南可以视为项目的第一个建设性质量保证措施。

其他的建设性质量保证活动大多是围绕具体产品进行的，它们有着不同的形成方式和规

范，如用例文档模板的规范可以要求在文档中给出用例规约对应的活动图，对于用例的命名方式，同样也可以进行约定。每个文档模板的制订都需要之前花费一定的时间和精力成本，然后确保文档的使用者对模板要求的所有内容都进行了填写和完善。所以，这里存在一个矛盾，如果模板过于强调正式和严格的质量要求，可能就会为此付出大量的成本或时间上的延迟，导致客户不满意。因此需要在面面俱到的模板和它们可能带来的阻碍之间折中考虑。一套示例性的项目文档具有很好的帮助作用，一方面提供了规范的模板和书写样例，融入了一定的技术经验；另一方面减轻了开发者的工作负担，使他们在此基础上根据具体的项目要求做必要的调整即可。

在程序设计中，还存在编码规范的文档，主要针对程序代码的风格做出规定。在实际项目中，通常借助工具来进行自动的检查和对照，因为人工检查的效率太低且容易出错。类似的工具有很多，例如，Java 开发中有一个编程规定是关于复杂布尔条件的格式化，在代码 11.20 中，未改进的代码的可读性较差，因为实际的执行指令没有与条件形成视觉上的分离；改进后的代码对布尔条件再次缩进，并将布尔运算的连接操作置于每行的开始，这样可以更容易地预览逻辑条件的概况。

代码 11.20　编码规范的示例

```
//未改进
if((condition1 && condition2)||(condition3 && condition4)||!(condition1
     && condition3))
   doSomething();
//改进后
if((condition1 && condition2)
   ||(condition3 && condition4)
   ||!(condition1 && condition3))
   doSomething();
```

对程序文本的格式化要求相对比较容易，关于程序风格规范的制订和应用就比较困难了。例如，“要尽量使用设计模式”只是陈述了一种要求，对其符合程度进行具体验证是比较困难的，而且还有不适用模式的例外情况，这就需要在测试中具体情况具体分析。

11.10　人工测试

很多功能性需求都是通过自动化测试进行验证的，同样，对于度量指标，也有相应的自动化工具支持。但是，对于有些产品项的要求却无法通过工具进行测试，如书写的错别字或语法等需求能够通过工具自动检查和纠错，但文档中与业务相关的含义和内容、可读性与模板的相符程度等，一般需要借助人工完成。

本节内容主要针对人工测试进行说明。人工测试有很多类型，可根据项目的实际需要进行选择。下面主要针对常见的人工测试方式进行说明，首先介绍一种正式的人工测试方法——审查(Inspection)；然后介绍非正式的、相对简短的人工测试方法——评审(Review)和走查(Walkthrough)。

图 11.14 使用了一个活动图对正式的审查流程进行了描述。审查活动的重点在于整个过程中对资源的计划，因为正式的分析过程需要耗费人员大量的时间和精力。

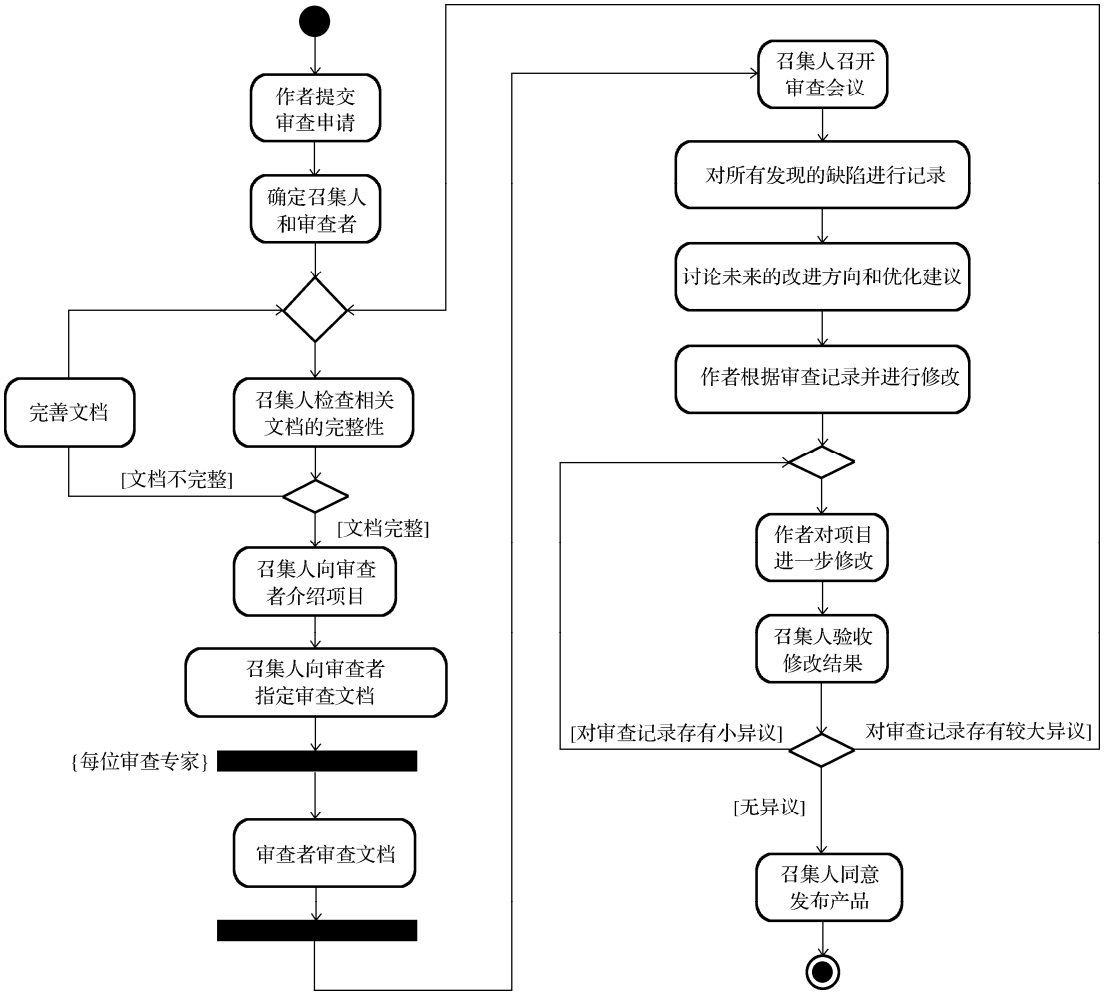


图 11.14　正式审查流程

另外，当作者完成文档后，应该可及时地得到专家的检查，这就需要在项目计划里制订对文档进行评审的安排并且预留出适当的延期缓冲时间。另外，需要建立项目评审专家组，其中通常包括一位来自质量保证部门的召集人，他只对整个审查过程的执行负责，对评审对象一般不持任何见解。对审查者的要求是能够从专业的角度对评审的产品项进行评估。召集人要做的第一步是组织审查者对文档的完整性进行检查，涉及的内容主要包括文档本身，还有产品项在创建时需要参考的其他资料或规范等，如需求中来源于客户的各种文字材料、使用的各种指南及文档的模板等。

召集人应了解每个审查者的职责和特长。对于大型产品的开发，审查者需要从不同的角度对产品进行分析，如客户的角度、基础构架的角度及开发的角度等。不同的审查者在各自擅长的角度上更富有敏锐度，而且也能够加快处理的速度。每个审查者接收到待评审的文档后开始独立地检查，这个过程中每个新发现的缺陷应被准确地记录，包括其来源、种类等。

召集人在收到和整理所有审查者的评审回执后，需要召开一次审查会议。在会议上将对所有发现的缺陷做统一的讨论和记录，若一个缺陷由多个审查者发现，则对其的描述需要综合几个审查者的意见重新进行整理；若审查者发现的缺陷是单独的，则需要由其他审查者一起进一步确认这是否是一个缺陷。审查会议应避免拖沓和冗长，因此会上着重对缺陷进行讨论，而不要过多讨论它们的解决方案，但可以给出一些对作者日后改进的建议。另外，还可以就本次审查工作做一下总结，如审查的形式及时间等是否合适等，为开发过程的改进提供进一步的参考意见。

产品作者通常不参加审查会议，他在会议结束后会得到一份会议记录，然后据此对产品进行修改和优化。最后由召集人负责检查各需要修改的项是否得到了处理，并做出是否允许该产品发布的决定。若召集人认为修改过于简单或者对产品做了过大的修改，则需要再次召集一次新的审查会议。

评审与审查比较起来主要的差别在于准备工作没有那么正式，同样需要所有评审专家对产品进行通篇的审阅，然后将发现的问题进行标注并给出评价意见。作者出席评审会议，可以针对专家的疑问进行回答，但一般不主动提出问题。作者在会后获得会议中所有评审专家的评审意见和他们的评论。

走查中作者作为主持人主持整个讨论过程，作为会议的主导，介绍自己的工作产品。其他参与者听取作者的讲解并提出一些批判的问题或改进的建议。为使讨论更有针对性，每个参与者也应该在之前对产品进行一定的了解和分析，这对于产品质量的提高也会起到积极作用。

人工测试要求至少三位人员参与，因此会带来直接成本。对于较为严格的评审过程，成本上有至少20%的增加。但对于文本的分析和检查，尤其是关于业务含义等语义内容，只有通过人工评审才能进行，因此在规模较大的项目开发中进行人工测试是很必要的。

除了增加成本，人工测试过程可能还会带来一些风险。例如，如果太多的人参与进来，就可能造成彼此协调沟通成本的大幅增加，从而大大降低效率。因此，在各种评审过程中只挑选那些有技术资格的人参加。另外，由于人工评审过程涉及对作者工作相对直接甚至针锋相对的评价，因此可能会产生一些无法预期的冲突，要避免对作者工作之外的评论并且将评审工作视为对工作改进的机会而不是发泄不满的途径，所以审查对象的作者不应作为评审团队中的成员参与进来。

对于一些在空间上分布的项目开发，评审的组织方式可以有其他的考虑：如借助 Web 方式提供一个共享的信息平台，通过 Wiki 等管理工具对评审的文档进行管理，为这些文档指定某种状态及评审终止的时间等。评审专家则可以打破地域的限制通过网络更加方便地对产品进行评价，召集人也可以借助现代工具实现对信息的收集。

11.11 习题

1．图 11.15 所示的流程图描述了某子程序的处理流程，现要求用白盒测试方法对子程序进行测试，并回答以下问题。

(1)满足条件覆盖的测试数据集，是否一定能满足分支覆盖？请举例。

(2)给出满足多条件组合覆盖的最小测试用例组。

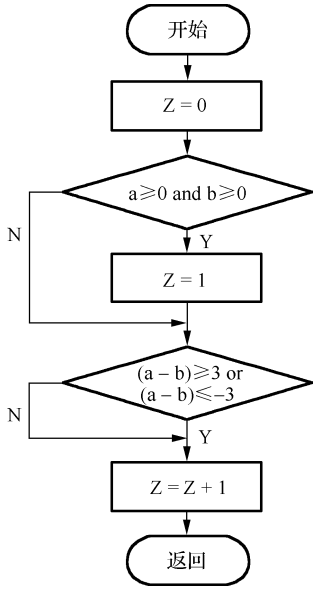


图 11.15　流程图

2．阅读代码 11.21，回答下列问题。

(1)绘制该代码对应的程序流程图；

(2)绘制该代码对应的控制流图；

(3)计算环形复杂度；

(4)使用条件覆盖标准设计该程序的测试用例。

代码 11.21　习题 2 代码

```
Input n;                //输入数组大小
Input List;             //从小到大输入 n 元有序数组
Input Item;             //输入待查找项
Start = 0;
Finish = n-1;
Flag = -1;
while(Finish - Start > 1 && Flag == -1){
    i = (Start + Finish)/2;
    if (List(i) == Item) Flag = 1;
    else if (List(i) < Item) Start = i + 1;
    else Finish = i - 1;
}
if(Flag == -1){
    if (List(Start) == Item) Flag = 1;
    else if (List(Finish) == Item) Flag = 1;
    else Flag = 0;
}
Output(Flag);
```

3．某函数完成教师课时津贴标准的计算。其输入参数有两个：教师职称及是否为外聘教师；输出为该教师的津贴标准。具体如下。

本校专职教师每课时津贴费：教授 50 元，副教授 40 元，讲师 30 元，助教 20 元；外聘兼职教师每课时津贴费：教授 50 元，副教授 50 元，讲师 30 元，助教 30 元。

使用等价类分析方法给出该函数的弱等价类测试用例。

4．图 11.16 所示是某杂志社稿件处理系统中稿件类(Article)的状态图，根据该图回答问题。

(1)给出 Article 类的定义，包括属性和方法。

(2)列出所有需要测试的类状态。

(3)列出所有需要测试状态的转换。

(4)从初始态开始(各属性置为空值)，一篇名为 *A Good Paper* 的稿件投稿，由于符合杂志领域范围，通过初审，然后送外审，外审专家认为质量不错，但需要继续修改，论文修改后达到要求，被录用。为以上场景开发一个测试驱动类，编写代码。

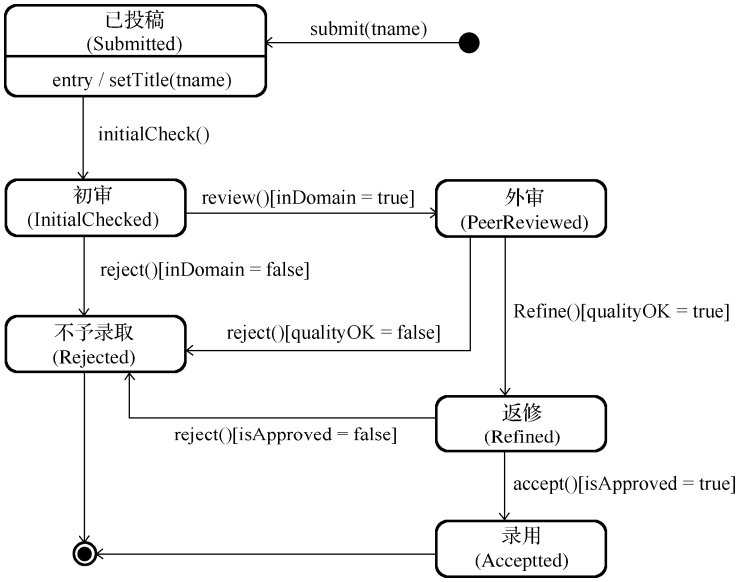


图 11.16　稿件类的状态图

第 12 章　软件项目级管理

前面的章节主要是从工程化的角度阐述了软件开发方法，这些方法大多基于实践中的经验，采用分而治之或逐步求精等手段来解决问题。许多采用这些面向对象方法并成功完成项目开发的团队在实践中逐渐意识到，对于大型项目的开发，除了技术上的“Know-How”是项目成功的基础，还存在另外一些能够左右项目成败的因素，它们对于软件产品质量也有深远的影响，这就是软件开发的管理技术和方法。

软件的管理可以分为两个层面，一是项目级管理，二是组织级管理。本章主要阐述项目级管理的相关内容，主要包括软件配置管理、软件计划、软件质量保证、风险管理、人员与沟通等方面，并在每个方面详细描述相关的技术和方法。

首先，软件开发者在一起进行团队开发，为了保证工作的一致性和高效性，需要软件配置管理的支持。其中，版本管理和构建管理是两个重要的活动。版本管理提供了一整套机制，使得软件开发者高效地进行团队开发，从技术和管理的角度明确地分工并确认各自的职责，保证每个人的工作都是基于最新的版本并且能够方便地分发开发成果。构建管理提供对系统自动生成的机制，确保每个开发者的开发成果可以正确、高效、自动地测试、集成甚至部署。

其次，从管理的角度来理解项目开发涉及的核心就是项目计划，项目计划包含的主要内容是软件规模、成本、进度等，形成可执行的计划书，同时按照计划，监控项目的实施，并对项目产生的偏差给予纠正。

最后，本章将介绍软件质量保证的相关内容，建立项目开发过程中风险管理的理念和基本知识。本章还讨论项目管理中的人员与沟通，这也是项目成功的重要因素之一，具体包括项目成员及社交层面上的团队合作，并说明项目成员如何通过一些交流的原则进行更好的沟通和合作。

12.1　软件配置管理

软件配置管理(Software Configuration Management，SCM)是一种标识、组织和控制修改的技术，贯穿于整个软件生命周期，包括的主要内容在图 12.1 中进行了概括。其中的核心部分是版本(Version)管理，由于提供了对所有产品项的统一版本管理，因此如果缺少它，其他内容将很难有效进行。构建(Build)管理能够提供产品之间依赖的路线，并按此路线对产品的生成进行实现。

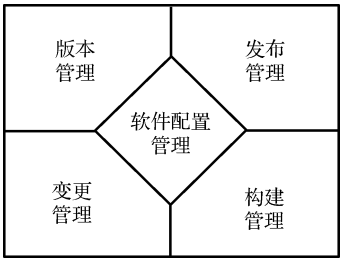


图 12.1　软件配置管理

发布管理(Release)的主要作用是协调在合适的时间对合适的用户交付合适产品，无论是对现有产品的扩展，还是

对缺陷的修正，都需要在版本系统的支持下保证交付的期限，尤其是在分布式的开发环境中。发布管理的必要性及作用包括以下几个方面。

(1)软件资源、软件开发过程及开发者的分散化，导致软件发布管理的复杂化。例如，在全球化协作的环境下开发，开发者分布在全球各个不同的位置，每天都有不同的人对软件进行更新，如果没有统一的发布管理，任何一个环节出现问题，都会导致复杂度急剧上升，日积月累，项目的交付日期就会没有保障，因此必须要有一个统一的环节对相关方面进行管理和控制。

(2)软件开发不是一蹴而就的过程。软件发布后，客户会对软件提出修改或者升级意见，并且为了抢占市场和抓住有利时机，很多软件不可能将所有需求全部实现再发布，而是先发布包含某些功能的版本，然后根据市场的反馈和相关决策进行后续开发。这样，为了保证产品的长期成功，需要统一的过程来管理这些发布和变更。

发布管理是对项目管理的一个有效补充。软件的持续变更和集成需要占有资源来保证软件开发流程。发布管理比起项目管理是在较细致的技术层面关注各种资源的协调，以保证开发过程顺利进行。

另外一个部分是变更管理(Management of Change，MoC)。软件生存周期内全部的软件配置是软件产品的真正代表，必须使其保持精确。软件过程中某一阶段的变更，均会引起软件配置的变更，这种变更必须严格加以控制和管理，保持修改信息，并把精确、清晰的信息传递到软件过程的下一步骤中。软件变更管理包括建立控制点和建立报告与审查制度，还包括对用户的确认及使其随时掌握变更的进度及细节。下面对版本管理和构建管理进行详细说明。

12.1.1 版本管理

版本管理主要包括两个方面的工作：一要规范化不同开发者之间的合作方式，必须保证一个人的工作不会被其他人意外地覆盖；二是要确保每个人的工作内容是当前需要的版本，而且能够为后续开发提供基础。

图 12.2 描述了一个开发者在软件开发过程中通过逐步对自己的工作内容进行优化的过程，是一种累进式的开发过程。开发过程中可能随时都有一些新的想法和实现，但随后又被抛弃掉，这就需要有一种机制，能够便于回到之前工作的某个状态，或者辅助开发者在需要的版本(快照，Snapshot)之间进行切换。

为了达到这些要求，需要将项目开发的历史版本管理起来。图 12.3 描述的就是一个项目的构成中每个开发包的演化历史，也反映了整个项目开发的版本过程。整个系统由三个主要的功能模块构成，每个模块的开发历史由 V1～V*n* 表示。在演化历史中，可能存在一些中间过渡的版本，它们并没有被采用到迭代的发布版本中，如模块 2 中的 V3。也可能会存在一些由外部施加的变更请求，如模块 3 中 V5 到 V6 的变化。整个过程中，重要的是开发者能够清晰地知晓哪些版本包含有哪些功能实现，及如何准确地获取到该版本，而且每个开发者在与其他人的工作一起联编时，还需要知道他们的工作版本和状态。这些开发过程的辅助工作都是由版本管理进行支持的，目前市面上也有很多类似的工具可供选择，如 SVN 和 Git，它们在实际项目中都有着广泛应用。

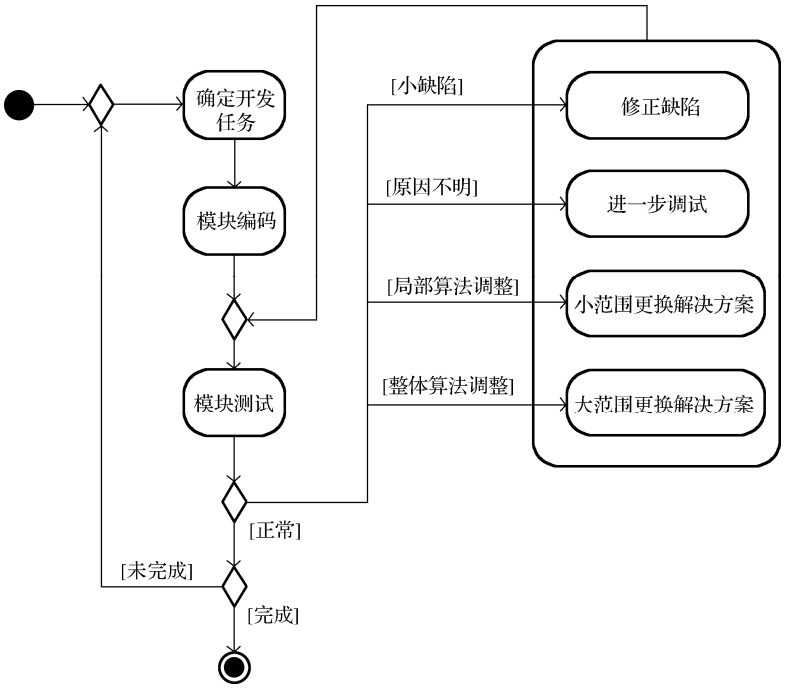


图 12.2　累进式的开发过程

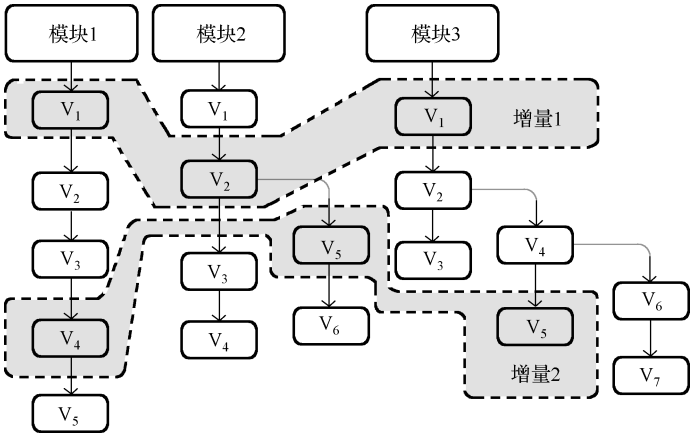


图 12.3　项目的版本演化历史

版本管理的核心工作是对项目软件或者项目文档的管理，把存储所有项目内容的数据库称为版本仓库(Repository)。版本仓库可以理解为一个存储着所有开发历史的数据库，与通常意义的数据库不同，它一般是在现有文件系统上的高效实现。

版本仓库的主要任务是为开发者提供整个项目从初始版本到当前版本的所有历史内容，并提供新版本项的创建。为此，版本管理系统在内部组织上采用了一套独立于操作系统的管理方式，一般存在一个管理员用户，主要负责管理版本仓库的创建、文档组织结构的初始化、备份的管理及版本技术方面的支持。所有的开发者作为版本管理系统的用户进行注册并被赋予一定权限，此权限与操作系统权限相互独立，如只允许项目组中的人员编辑或创建文件。

所有纳入版本仓库进行管理的各种软件资产统称为软件配置项，包括各种文档、数据及

代码等，每个配置项都有一个独立的版本历史，称为代码线(Codeline)。这些配置项在其生命周期的特定时间点上通过正式评审进入正式受控的状态，把这些软件配置项对应的版本、配置文件及依赖的外部库等的集合称为基线(Baseline)。作为版本管理的基础，基线可理解为一个项目完整、稳定的版本，基线为后续开发活动提供了信息的稳定性和一致性。版本管理中另外一个概念是主线，它是指一组顺序的基线构成的集合，代表了系统的演变路线。

图 12.4 描述了开发者使用的一种常见的二区结构的版本管理系统的典型工作过程。开发者并不是直接在服务器端版本库中的最新版本上工作，他们需要将最新版本复制到本地工作区，这样服务器上的最新版本就会一直留存。这个通过版本管理系统的复制动作称为检出(Check Out)，检出后会在用户本地工作空间创建一份与服务器版本一模一样的副本，然后开发者就可以在这份副本上持续工作，同时版本系统也可获知这个开发者工作的当前版本。如果开发者结束本次工作，需要将其工作成果反向复制回版本系统中，把这个动作称为检入(Check In)或提交(Commit)。检入操作会在仓库中延续对应的版本树结构(代码线)，并且使其版本号在原来的基础上做一次升级。在检入的同时，系统还会要求开发者给出简短的对本次修改缘由的描述。

版本管理系统除了能够对最新的发行软件进行版本的管理，还可以为开发者提供对所有历史版本的浏览和检出功能。在图 12.3 中的版本树，不同的版本会对应不同的用户，开发周期较长的项目，就需要将整个版本树都纳入版本管理系统中。版本项的检入和检出是版本管理系统提供的两个基本操作，若多个开发者同时修改同一个项目的同一个文件，则经常会产生冲突，这在图 12.5 中进行了描述。首先开发者 Eric 检出了文件 D 的版本 *n* 并进行编辑，然后 Kate 也检出了同一版本的文件，在做了简单的编辑后检入了文件，然后 Eric 结束编辑并执行检入操作，这个过程如果版本系统不加任何警告和控制，刚才 Kate 所做的工作很有可能会被覆盖或丢失。解决这种冲突问题有两种方法，不同的版本管理系统可能使用不同的方法。

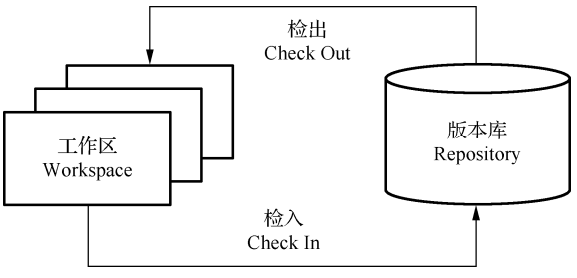


图 12.4　二区结构的版本管理系统

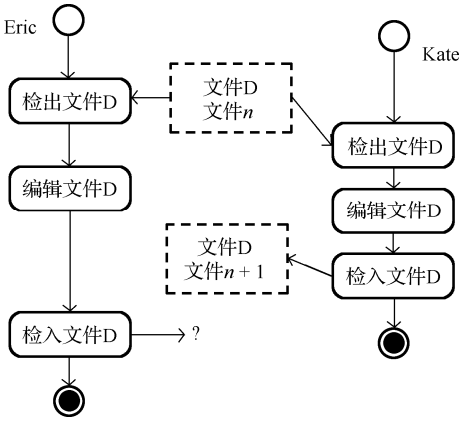


图 12.5　版本管理系统中的冲突

第一种方法称为悲观的方法，在这种方法下，Kate 无法检出该文件。其一般原理是：第一个检出文件的人将会拥有对该文件的排他锁，当 Kate 试图执行检出已经被 Eric 检出并正在编辑的文件时，将会收到检出失败的提示。称其为“悲观”是因为每次检出都假设能够产生一个潜在的冲突并做好加锁的预防措施。这种方法的好处在于能够保证一个文件同一时间只由一个人进行编辑，并且不会产生任何冲突，但存在的问题有时也很严重，并且无法接受。

例如，当 Eric 处理该文件时花费了较长的时间，一种极端的情况是 Eric 检出文件后忘记检入，然后去旅行了一段时间，Eric 旅行期间，Kate 即使想做一个与 Eric 工作无关的小修改，也不得不等待。

第二种方法称为乐观的方法，Eric 和 Kate 可同时对文件进行编辑。当 Eric 最后检入该文件时，系统会自动进行检查，然后提示 Eric 存在冲突无法成功检入，因为他的当前版本为 *n*，而版本仓库中该文件的版本为 *n*+1。在这种情况下，Eric 需要再次从服务器检出(Update)最新版本，并在他本地由版本管理系统识别两者的修改发生在同一文件的具体位置，然后尝试自动将二者的结果进行合并，否则 Eric 必须手动解决冲突，这个过程通常需要借助一种自动进行文件内容比对的辅助工具软件，通常称其为 Diff 工具。Diff 工具也属于版本管理工具之一，它的工作方式是按行进行文本内容的比较，分辨出哪些行被删除、修改或新增了，图 12.6 给出了一个 Diff 工具的示例。Eric 通过该工具的支持能够方便地进行变化的合并并解决无法自动合并的冲突，最后通过版本管理工具检入版本 *n*+2。

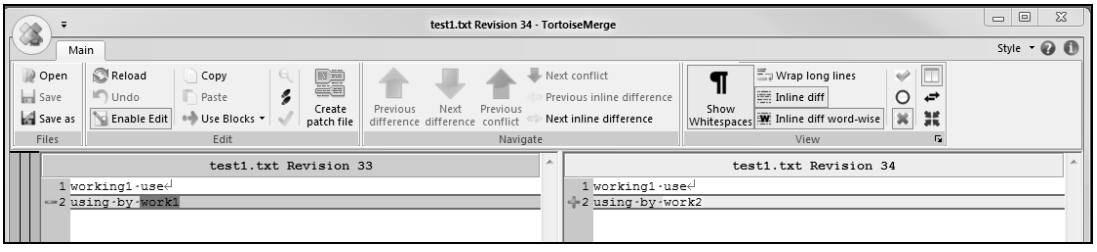


图 12.6　Diff 工具示例

图 12.4 介绍了常见的二区结构的版本管理系统，如 SVN 采用的就是这种结构，另外还存在三区的结构，如 Git 使用的就是三区结构。接下来，我们简单介绍一下 Git 所采用的这种三区结构。三区即工作区、暂存区和版本库，与二区结构不同的是中间增加了一个暂存区，如图 12.7 所示。由于 Git 是一个分布式的版本管理工具，因此，版本库又可以继续细化，分为本地版本库和远程版本库。

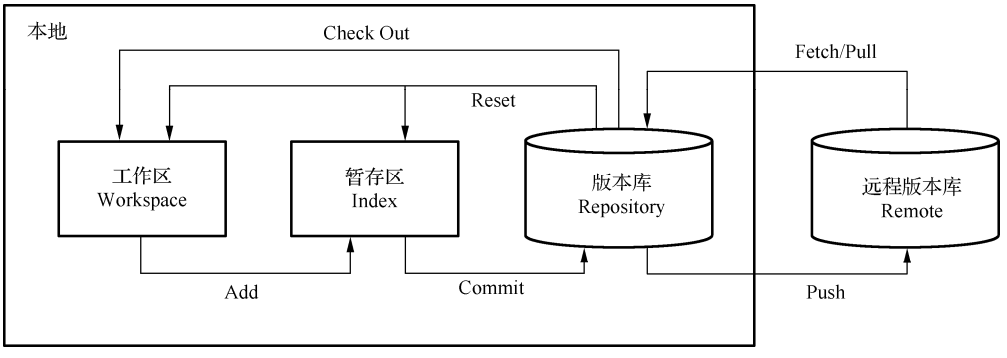


图 12.7　三区结构的版本管理系统

此时工作区直接和暂存区交换文件，而暂存区和版本库交换文件。也就是说，用户在工作区完成文件修改，通过 Add 将文件提交到暂存区，暂存区再通过 Commit 将文件提交到版本库中。当用户再次修改文件时，重复这个过程。Git 是一个分布式的版本控制工具，具体体现在 Git 同时拥有本地版本库和远程版本库，在用户将文件提交到本地版本库之后，还可以通

过 Push 命令进一步提交到远程版本库上。同样，更新文件时，需要从远程版本库上 Fetch/Pull 文件到本地版本库，再检出到本地工作区。这个过程使得用户在其本地能够持续连贯地进行版本的演化，不必受到网络连接的限制。

版本管理系统与其他工具联合使用会发挥出更大的价值，提供更多的选择，以使开发保持在一个高质量的层次。例如，在检入时与设置的测试工具 JUnit 联用(只有 JUnit 测试后没有错误发生时，才允许检入的执行)。

12.1.2 构建管理

构建管理系统的主要任务是描述最终软件产品的结构和生成过程。对于小型的开发项目，构建系统的作用是调用编译器，然后根据编程语言的不同，使用不同的链接器生成可执行文件并执行。即使是小型的项目，这些基本步骤的成功执行也与某些具体的条件相关，如编译器的位置、在编译目录中创建文件的权限等。另外，可能还存在一些其他约束条件，如与指定目录的类库和构件的联编等。对于大型的开发项目，在构建的构成中还要确保只有那些被修改的部分，才会进行重新编译和链接。除了编译这项主要的工作，围绕文件的处理还有其他重要的任务，如目标软件在生成后应指定在系统中安装的具体位置等。

上述这些操作可以借助命令行等工具进行手工执行，如 Shell 或 Dos 等。为减轻工作负担，首先考虑可以将 Build 信息写入一个 Shell 脚本或批处理文件中自动执行，这种方式使用起来比较简单、顺畅，但当开发较为大型的系统时，往往需要一个可维护的开发环境，这时需要引入专门的 Build 管理工具，如 C 或 C++常用的 Make 工具和 Java 的 Ant 工具。

更大型的项目可引入持续集成来进行更为复杂的构建管理，持续集成的目的就是让产品可以快速迭代，同时还能保持高质量。它的核心措施是，代码集成到基线之前，必须通过自动化测试。除了高效地生成系统，持续集成还能够自动化地部署系统，因此持续集成的作用其实并不能消除软件缺陷，而是让它们非常容易被发现和改正，并使得系统的构建变得高效和自动化，图 12.8 为持续集成过程的示意图。

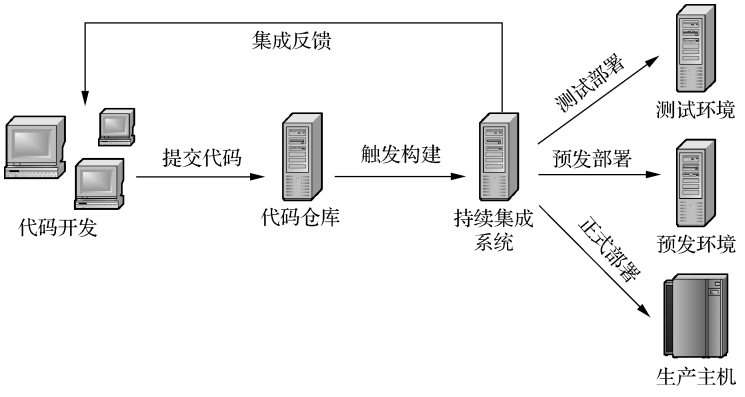


图 12.8　持续集成过程

这里不对持续集成的内容进行展开，主要通过 Ant 工具针对以上内容在基本概念的层面上进行具体说明。在构建管理的应用领域，Ant 或者 Make 工具的两个主要作用是：

(1)提供边界条件的管理，如系统配置及其他相关变量；

(2)命令链的执行管理，其描述了从某些对象出发构建新对象的过程及结果等。

第(2)个作用中，典型的活动是将 Java 源文件在编译器的作用下生成 class 结果文件，并输出到某个特定的目录中。命令链描述了一个目标及实现此目标需要具备的前提条件，在 Ant 中同样具有命令链的概念和相应的描述方式。另外，由于每个目标可以由更小的子目标构成，因此整个命令链构成了一种树状的层次结构。代码 12.1 展示出了这样的结构。

代码 12.1　Ant 的构建配置示例

```
<?xml version="1.0"?>
<project name="AntExample" default="default">
   <description> Demo with ant </description>
   <property name="srcdir" location="src/AntExample/test"/>
   <property name="builddir" location="build"/>
   <property name="destdir" location="dest"/>
   <target name="default" depends="clean,compile,pack,execute"
               description="do all">
      <echo message="in default"/>
   </target>
   <target name="clean" description="clean up">
      <delete dir="${builddir}"/>
      <delete dir="${destdir}"/>
      <echo message="in clean"/>
   </target>
   <target name="start" description="initialization">
      <tstamp/>
      <mkdir dir="${builddir}"/>
      <mkdir dir="${destdir}"/>
      <echo message="in start"/>
   </target>
   <target name="compile" depends="start" description="compile">
      <javac srcdir="${srcdi r}" destdir="${builddir}" classpath="."
                  Debug="on"> </javac>
      <echo message="in compile"/>
   </target>
   <target name="pack" depends="compile" description="package">
      <jar destfile="${destdir}/packed${DSTAMP}.jar" basedir=
                  "${builddir}">
         <manifest>
            <attribute name="Main-Class" value="AntExample.test.
                        XStarter"/>
         </manifest>
      </jar>
      <echo message="in pack"/>
   </target>
   <target name="execute" depends="pack" description="execution">
      <exec dir="${destdir}" executable="cmd.exe" os="Windows XP"
               spawn="true">
```

```
            <arg line="/C java -jar packed${DSTAMP}.jar"/>
        </exec>
        <echo message="in execute"/>
    </target>
</project>
```

这个脚本以 XML 的格式描述了一个 Build 文件的构成，并能够由 Ant 直接进行解释和处理。脚本的第 2 行指定了一个默认目标 default，表示当在参数中未指定具体目标时，默认针对第 7 行的目标进行生成。文件中的 property 元素表示对某个变量的定义和赋值，应该总是给出其名字和对应的值。对变量 var 值的引用使用${var}的形式，路径分隔符在 Windows 中使用的是“/”，而在 UNIX 中使用的是“\”，在 Ant 的构建文件中则两者都可以使用，提升了可移植性，为跨平台的使用提供了便利。

脚本文件的重要组成部分是目标元素 target，其总是存在一个 depends 属性，描述一个依赖项的列表。Ant 在处理过程中会分析既定目标所依赖的内容，然后逐步实现这些目标。当然，其中不允许循环依赖的情况。

每个目标的实际执行是由 Ant 的任务部分完成。它们包含在 target 元素中，是一些可以附加参数的预定义命令，在每个目标的描述中顺序地执行。

脚本中给出了一个简单的任务示例<echo message="Text"/>，如第 8 行，其作用是在任务进行时在控制台打印出消息的文本内容。子目标 clean 和 start 的作用是对指定文件和目录进行清理和创建。子目标 compile 和 pack 的作用是对 Java 文件编译并且输出结果到指定目录，然后使用 Java 通用工具 jar 对结果打包，最后生成能够执行的 jar 包。pack 中使用了一个变量 DSTAMP，它在目标 start 中通过命令<tstamp/>被初始化，该变量的值记录了每次运行该脚本的开始时间，这个时间可以作为一个唯一值为不同的文件产生它们的名字，如 packed20180210.jar。子目标 execute 的作用是直接调用操作系统的命令，如运行打包后的程序。

针对 Java 的开发，Ant 已经定义了很多类似的 Ant 任务，而且在一些现有的开发工具中 Ant 允许将功能嵌入其中，从而构成一种对 Ant“无缝”的使用方式。相对来说，通过程序来构建自己的 Ant 任务也是较为容易的，因为 Ant 提供了一个 Java 框架方便开发者对其进行扩展，这对于构建定制的软件开发环境是非常方便的，可以做到细粒度的工具集成。除了 Ant，还有一些更为强大和复杂的构建系统，如 Maven、Gradle 等，它们经常用在大型的项目开发中。

12.2 项目管理

除了开发的方法、技术及开发环境，项目管理是项目成功与否的又一重要因素。与技术实现的关注点不同，项目管理主要关注组织和管理层面的内容。

项目计划实际上是对项目规模、工作量、成本、进度等方面的估算，同时对人员、时间、计算机资源等各类资源统筹安排，起到非常重要的作用。准确且系统地评估出项目的实际成本几乎是不可能的，将业务领域、使用的工具、参与开发的人员等因素统筹考虑并集中到一个最终的公式中也难以做到。本节的主要目的是使得每个相关的人员能够理解项目成本估算的方法并建立起估算过程的概念，使得估算趋于准确，同时基于成本的计算，明确项目的总体报价并最终将该报价写入合同作为一项法律的依据。

项目管理覆盖了较大的知识范围，而且内容庞杂，因此本节主要选取项目管理中与技术和开发者比较密切的一部分内容进行介绍。主要目的是使得相关人员懂得项目管理在整个软件开发中的原理和必要性。

12.2.1 项目计划与工作分解

软件项目往往来源于某个想法，这个想法逐渐由客户或者企业成熟化并付诸实施。软件开发项目组根据对项目的理解和业务分析，考虑各种限制或条件的影响，并在此基础上与客户商讨开发价格，根据市场及目前的财务状况，评估出是否可以盈利及是否具有足够的资源来支持项目开发。

项目经理在项目中一般处于核心的领导地位，他负责项目在整体上的计划、进度控制等工作，要随时解决项目开发过程的各种问题，应对外部客户和团队内部开发相关的事项。当然，这不是项目经理需要具有的全部能力，他需要具备综合的能力和素质，如周全的斡旋能力和快速的决策能力，而且，为了能够与开发团队沟通，项目经理应当具有相当的技术基础和专业素质；他应该对业务具有较深入的理解，最好能够达到领域专家的水平。当然，在一些较大型的软件系统开发中，项目经理的任务通常被分解为以管理为主和以技术为主两大部分，分别由不同的人担任，如项目领导(Project Leader)和技术领导(Technical Leader)。

在项目计划开始时，要决定哪些任务需要完成。这个工作可以通过工作分解结构(Work Breakdown Structure，WBS)的机制展开，将任务按照层次的结构由上到下逐步进行分解，图 12.9 给出了一个 WBS 的例子。在分解的过程中值得注意的是，子任务间的工作内容应尽量做到无重叠或较少的重叠。图 12.9 所示的例子中还存在一些跨任务的工作，如质量保证、项目管理及针对用例的技术工作。项目还可以有其他的分解方式，如根据目标软件的构成进行分解，也是在常用的方式。在图 12.9 中的 WBS 结构中，每项工作任务同时也给出了对应的工作量，这里使用单位“人天(PD)”进行表示，全部的工作量是所有指定工作时间的合计。当然，这里的每一项只是在项目初始对工作量的估算，与实际情况越符合越好。对工作量评估的几种方法将在以下章节中说明，这里假定已经得到了工作量的一个合理评估，每个工作包应存在两个评估值——期望的工作量和为潜在问题预留的缓冲量。

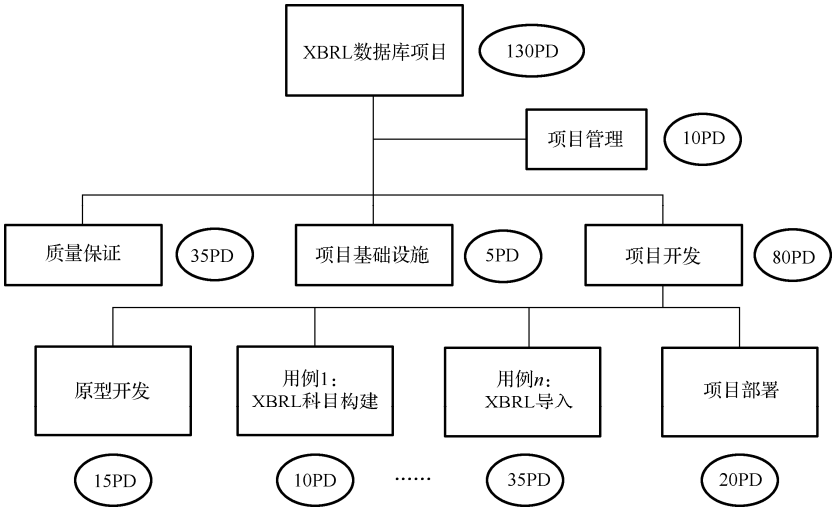


图 12.9　WBS 示例

12.2.2 软件规模估算

传统的软件规模估算方法是一种基于分解的方法，项目可以按照 WBS 的方式分解为子项目。然后由一个或多个专家基于他们的经验对各子项目的工作量分别进行评估，这个过程中的主要评估方式采用“类比”的方法，即参照以往相似项目的实际工作量得出一个估算的结果。如果多个专家进行不同的独立评估，那么可以通过一个评估会议对这些结果进行综合，然后产生一个最终的结果。如果某个专家的结果与其他结果有较大的偏差，那么该专家应该解释在该任务中是否真实存在着某些导致成本增加的潜在原因。这些讨论通常会揭示一些隐藏在任务中并且容易被忽视的因素，从而对项目产生新的理解。

成本估算与风险管理是密切相关的，往往在做第一次估算时就会考虑和揭示出与项目开发相关的某些风险并记录它们。风险会带来成本的增加，因此，用一个系统化的方法来评估风险带来的成本也是必要的。在实践中，一般采用的方法是先给出成本的一个估计值，然后根据风险的大小在这个估计值的基础上预留出风险缓冲的范围，如通常会预留 30%～50%的风险成本。

另外一种系统化的评估方法是功能点分析。它是一项可以进行认证的评估技术，其评估过程如图 12.10 所示，接下来使用具体的例子进行解释。这里假定要实现的是一个较小的员工差旅管理系统，主要实现对员工出差报销的管理。该系统面向的是中小型公司，实现上采用的是 Java 语言，由于数据量并不是很大，所以数据存储方案没有采用数据库的方式，而是将数据直接存储在 XML 文件中，同时也使得对数据的使用更为简单和轻量化。交互界面中只包括期望的基础功能，主要的功能包括员工的添加、员工资料的更新、员工的删除、行程安排(录入)、行程单据录入、行程审批、行程费用计算、行程费用报销等。

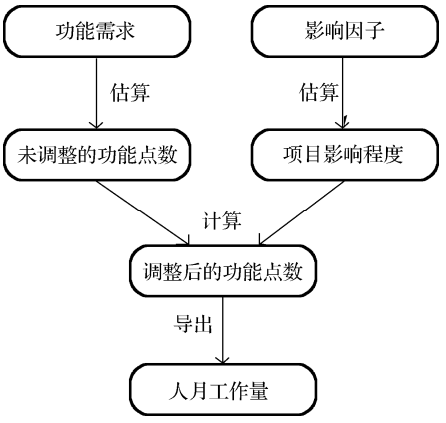


图 12.10　功能点分析的过程

更为详细的功能描述在此处被省略了，因为成本估算往往都从较少的信息开始。当然，系统需求分析的过程进行得越细致，估算的结果就越准确。在功能点分析中，需要准确地理解当前所有的用户功能，并尽可能地将每个功能归到以下 5 种任务类型之中。

(1)内部逻辑文件(ILF，内部数据)：在待开发系统内部处理的数据，如开发的类本身。

(2)外部结构文件(EIF，引用数据)：从待开发系统的外部引入并进行处理的数据。

(3)外部输入(EI，输入)：来自待开发系统外部的输入，并由此对数据展开处理，如数据以某种约定格式(输入掩码)从系统外部输入。

(4)外部输出(EO，输出)：在待开发系统中，实现业务计算结果的外部输出，例如数据以某种输出格式(输出掩码)或对其他系统的错误消息进行输出。

(5)外部查询(EQ，查询)：从外部系统发出的对数据信息的查询，不包括其他需要的附加计算。

然后，在功能点类别的基础上，对功能的复杂度进行考虑，可以简单地分为容易、中等

和复杂三个级别，如输入类别对应三个级别的可选值可以分别为“3-4-6”。在功能点相关的文献中存在不同的评估方法，它们对于复杂度的度量也有不同的表达。

接下来的工作是基于对各个功能点的估算结果利用一个模板计算出未调整(无权重)的功能点分值。每个功能点按照功能类别和难易程度指定对应的值，这些值又与其出现的频率相乘。表 12.1 给出了一个示例性的评估计算，在“员工的删除”一行中的值 7 表示其含有对数据的一个简单操作，“员工的添加”一行中的“1×6”，表示这是较复杂的输入功能，可以将对应的功能点值进行倍值计算。所以，深入理解和提炼用户需求对准确地确定出功能点所属的类别和分值具有很大的帮助。

表 12.1　未调整的功能点数计算

任务需求	输入 3–4–6	查询 3–4–6	输出 4–5–7	内部数据 7–10–15	外部文件 5–7–10	Σ
员工的添加	1×6	—	—	15	5	26
员工的更新	4	3	5	7	—	19
员工的删除	—	—	—	7	—	7
行程安排	4	—	4	7	—	15
单据录入	6	—	4	7	—	17
审批	—	3	4	—	—	7
费用计算	—	4	4	—	5	13
费用报销	—	—	4	—	5	9
合计						113

无权重的功能点力求对用户所有的功能性需求进行一致的评价，但在大多数的开发中往往很多因素会影响开发的难度，即项目的约束或边界条件的影响。需要首先将这些因素进行整理和归类，作为每个功能点的权值纳入计算，取值范围可定义为 0～5。例如，对于适应性的影响因素，0 表示只进行一次简单开发，5 表示需要长期的开发和维护，后期可能还要在此基础上进行扩展。估算出的每个影响因子按照权重计算公式进行综合，然后作用于先前估算出的无权重功能点值，进而产生最终的有权重的功能点值，该例子调整后的功能点数计算见表 12.2。

表 12.2　调整后的功能点数计算示例

未调整的功能点数(UFP)		113	备注
影响因子	与其他系统的交互(0～5)	0	无交互
	分布式数据的处理(0～5)	3	客户/服务器
	事务的高处理率(0～5)	1	较低的连续处理要求
	处理逻辑		
	计算复杂性(0～10)	2	简单计算
	控制复杂性(0～5)	2	中等要求
	出错处理(0～10)	3	数据一致性检查
	业务逻辑(0～5)	2	标准要求
	可重用性(0～5)	1	较低要求
	可移植性(0～5)	1	较低要求
	可维护性(0～5)	5	较高要求
综合影响合计(DI)		20	—
影响因子(TCF)= DI/100 + 0.7		0.9	—
调整后功能点数(FP)=未调整功能点数(UFP)* TCF		101.7	—

12.2.3 开发成本估算

最后，需要将带有权重的功能点评估分值转换为对工作量的评估，这个转换过程在一些文献中都有介绍，其中一种方法是通过功能点经验曲线，如图 12.11 所示。这条曲线针对不同的公司甚至不同的部门，曲线的形态都不一样。

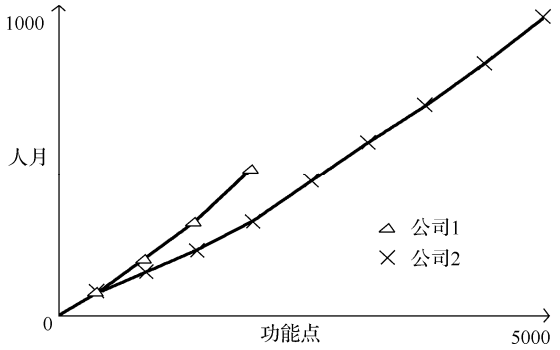


图 12.11 功能点经验曲线

以上估算方法的主要问题是在估算的过程中存在诸多较模糊的主观判断，可能会对最终结果产生较大的影响，但能通过后期的一些比对和反馈训练，使得评估的表现获得改善。另外，评估过程可以由有经验的人员多次取平均值，整体上的评估结果也要更好一些。可以尝试对已经结束项目进行训练性的评估，并将评估结果与实际结果进行比较，在数据中找出偏差趋势，用以对未来项目的评估结果进行修正。

利用功能点技术来对成本进行估算的另外一种方法是使用模型 CoCoMo（Constructive Cost Model），它是由 Berry Boehm 针对不同领域的实际项目进行估算和相关数据的持续跟踪整理的结果。这里介绍的模型是 CoCoMo II 模型，该模型包括三个子模型，分别是 Application Composition Model、Early Design Model 和 Post Architecture Model。表 12.3 总结了这三个子模型的作用。

表 12.3 CoCoMo II 的子模型

子模型	作　用
Application Composition Model	针对使用集成 CASE 工具进行快速应用开发的项目工作量的评估及计划调度
Early Design Model and Post Architecture Model	针对基础软件设施、重要应用及嵌入软件项目开发的工作量的评估及计划调度

CoCoMo 模型用来估算目标程序使用某种编程语言的代码行数，也可以使用未加权重的功能点技术将功能点转换成对应的代码行。CoCoMo 模型是模型的集合，根据项目的级别或种类选取其中不同的模型来使用，这里主要针对项目开始的早期设计模型进行介绍。CoCoMo 模型最大的作用在于提供工作量（Person Months，PM）估算的公式，并以此作为项目计划调度和优化的基础，公式的基本形式为：

$$\mathrm{PM} = A \times \mathrm{Size}^{E} \times \prod_{i=1}^{n} \mathrm{EM}_i$$

式中，Size 是指不含注释的程序长度，又称 KDSI（Kilo Thousands of Delivered Source Instructions），因此其基本的单位是千代码行。

公式中表示每个影响因子的变量 EM_i 为工作量系数，它们以乘积的形式线性地融入公式中。表 12.4 中针对早期设计模型，使用了 7(n=7)种可能的影响因子对模型特征及它们的取值进行指定。在功能点方法中，对未加权重的功能点值施加项目因素的影响，最大有 30%的修改空间，但在 CoCoMo 模型中，各种因素有更大的影响，如表 12.4 中的特征“人员能力”。

公式中的 A 是一个常数，其取值依赖于不同的业务领域，对结果也是线性的作用，如对于 Web 和军工、金融等应用，其取值介于 2.5 和 4 之间。

表 12.4　工作量系数 EM 的取值

影响因子	特低	很低	低	正常	高	很高	特高
可靠性及复杂性	0.73	0.81	0.98	1	1.30	1.74	2.38
可重用性	—	—	0.95	1	1.07	1.15	1.24
目标平台特殊性	—	—	0.87	1	1.29	1.81	2.61
人员能力	2.12	1.62	1.26	1	0.83	0.63	0.50
人员经验	1.59	1.33	1.12	1	0.87	0.71	0.62
开发环境	1.43	1.30	1.10	1	0.87	0.73	0.62
开发周期	—	1.43	1.14	1	1	1	

公式中的参数 E 以指数形式影响整个项目的成本，因此比其他参数的影响能力要更大一些。E 的取值可以参考以下的公式，其中的正则化因子 B=0.91。

$$E = B + 0.01 \times \sum_{j=1}^{5} \mathrm{SF}_j$$

在确定 E 值的公式中同样也存在几个待估计的比例系数 SF，其取值可参考表 12.5。总体上，CoCoMo 模型综合了对项目有较大影响的一系列边界条件，同时也提供了根据实际情况对影响因子进行调整的机会。另外，CoCoMo 模型实际的表现与评估者本身的经验也有直接的关系。

表 12.5　比例系数 SF 的取值

影响因子	很低	低	正常	高	很高	特高
有过类似的开发先例	6.20	4.96	3.72	2.48	1.24	0
开发的灵活程度	5.07	4.05	3.04	2.03	1.01	0
架构/风险方案	7.07	5.65	4.24	2.83	1.41	0
团队凝聚力	5.48	4.38	3.29	2.19	1.10	0
软件开发过程成熟度	7.80	7.80	4.68	3.12	1.56	0

从以上的分析中可以看出，利用功能点分析和 CoCoMo 模型方法进行成本估算的关键是评估者的经验，因此，可否由开发者主导评估，而不仅是参与评估过程？因为开发者参与实际项目的开发，与开发团队和客户经常探讨业务和技术的融合，对系统具有较好的理解。当然，如果评估的开发者有过完整的与新项目相似类型和约束的项目经验，会使评估的结果更准确。

由开发者主导的评估过程可以由需求分析中被识别出来的用例开始。每项任务根据表 12.6 给定的四类评估影响因子 A、B、C 和 D 分别在规定的取值范围内进行估算。四类因子的取值对最终的工作量有直接的影响，并且彼此被认为是独立的。因子 B 和 D 在某些情况下可能会混淆，因为非常关键的程序往往具有比较复杂运算逻辑。当然，情况也可能是相反

的，即逻辑计算非常简单，但是关键程度非常高。这里假定各因素间的关系是独立的，这有助于理解通过公式 A×B×C×D 计算总体工作量的合理性。

在评估过程中，首先对评估进行校准，即针对中等规模任务的 B 值，至少要由三位评估者参与，所有评估者理解这些任务并能够就它们的 B 值等于 4(中间值)的情况达成共识。然后以此为基础，每位评估者独立对所有余下的任务进行评估，之后召开评估会议，就大家所有的评估值进行讨论，及时记录发现的问题和风险。当然，这里只是给出工作量在任务间的相对关系，还没有得出工作量的具体值。

表 12.6　简化的分析评估影响因子

影响因子		描　　述
A(通过重用进行构建的可能性)	1	没有通过重用进行构建的可能性，所有内容都需要重新开发
	1.3	具有一定的通过重用构建的可能性，但需要识别出重用的内容和位置并进行适应的调整
	1.6	具有较高的通过重用构建的可能性，需要较少的调整，已有的文档对可重用的部分具有详细的说明和扩展方法
	1.9	大部分可通过重用进行构建，设计上也符合要求，不需要太多的调整
	3	全部可基于现有的成果进行构建，开发的主要工作是测试
B(任务规模，没有注释的代码行数)	1～8	对任务的规模进行估计，这里要求任何任务的规模不能超过最小任务规模的 8 倍，每个任务的给定值是相对值。若完全可通过重用构建，则 B=0
C(接口数目)	1	不依赖或较少依赖其他任务
	2	中等依赖其他任务
	3	重度依赖其他任务
D(在项目中的关键程度)	1	低度，当有问题发生不会导致整个项目的失败
	1.5	中度，当有问题发生功能性和客户满意度将会受到影响，但系统仍可用
	2	高度，当有问题发生将导致整个项目的失败

在接下来的步骤中，对一个已经结束的项目进行同样的独立评估并最终进行同样的评估会议。由于已结束项目的成本是已知的，因此可以通过如下方法评估新项目的成本。

(1)对每个新的任务 new_i，计算其 A×B×C×D 的值。

(2)将各个任务的 new_i 值相加，计算新项目的工作量分数：EP_new=new_1+…+new_n。

(3)对每个已完成任务 old_i，计算其 A×B×C×D 的值。

(4)将各个任务的 old_i 值相加，计算已完成项目的工作量分数：EP_old=old_1+…+old_m。

(5)假定 RP_old 为以“人天”为单位的已完成项目的实际工作量，计算工作量比例值：Value_RE=RP_old/EP_old。

(6)使用比例值 Value_RE 计算新项目的成本估算值为 EP_new×Value_RE，其中，每个单一任务的工作量为 new_i×Value_RE。

所有的评估者对最终的结果进行合理性检查，并根据情况进行必要的调整，而且可以基于新、旧项目的约束条件对它们进行比较。借助于功能点分析或者 CoCoMo 模型的评估特征，可分析出新、旧项目在条件约束上是否具有相似性，因为这些都能够在影响因子的取值上体现出来。

对以上评估过程的分析可知，每个影响因子的取值范围与计算的过程是不相关的，如果调整所有的取值范围都是 1～10，计算过程还是会以相同的方式进行。实际采用的取值区间对应该影响因子的重要程度，这个重要程度应与开发者的期望一致。

目前的方法都用来评估项目整体或者任务(功能)的成本，如果需要评估管理活动的成本，如项目管理或者质量保证等，一般是在总成本的基础上根据每个部分所占的比例进行估算。通常，随着项目规模的增加，实际的开发工作量所占的比例会逐渐下降，其中的管理成本、文档化的成本及质量保证的成本却在增加，它们各部分的比例关系如图 12.12 所示。

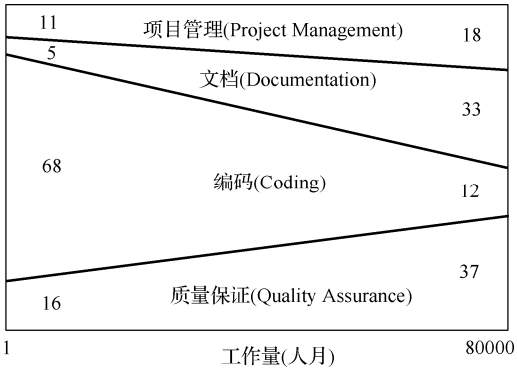
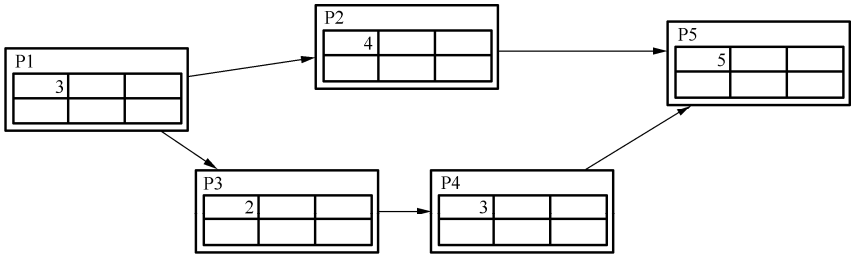


图 12.12　项目开发中各部分的成本比例

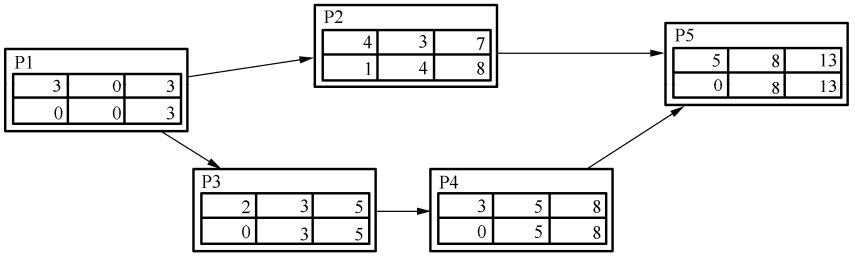
最后需要说明的是，评估需要在项目进行过程中随时比对和做必要调整，因此评估需要重复进行，CoCoMo 模型提供了精炼模型，用以对评估结果进行改进。对项目的估算成本及实际成本，在项目结束后都应记入项目开发的资料文档中，以便为其他项目的分析提供历史数据支持。

12.2.4　任务安排与工程网络图

为了更好地完善计划，除了需要合理地估算每个工作包的工作量，还需要能识别出它们彼此之间的依赖关系。具体来说，就是要理清工作任务的优先级和顺序性。



(a) 计算前



(b) 计算后

图 12.13　工程网络图示例

工作任务之间的依赖关系可以通过图形的方式进行展现，图 12.13 所示是一个工程网络图，其中子图(a)为给定依赖关系和每个任务的持续时间，子图(b)为计算之后的每个工作任务的时间安排和持续情况，假定时间单位为小时。工程网络图中的每个节点代表一个工作包，其信息包括 6 个部分，如表 12.7 所示。

表 12.7　工程网络节点详细信息

工作包名称		
持续时间	最早开始时间	最早结束时间
机动时间	最晚开始时间	最晚结束时间

工程网络图的计算分为两个阶段，分别是前向计算和后向计算。前向计算主要处理最早时间，包括最早开始时间和最早结束时间。如果该工作包是开始节点，那么最早开始时间为 0；否则最早开始时间等于前面工作包的最早结束时间(取大值)；最早结束时间等于该工作包的最早开始加上持续时间。后向计算时，如果该工作包是结束节点，那么最晚结束时间为其最早结束时间；否则最晚结束时间等于后面工作包最晚开始时间(取小值)；最晚开始时间等于最晚结束时间减去持续时间。

网络计划时间参数计算的目的在于通过计算各项工作的时间参数，确定网络计划的关键工作、关键线路和计划工期，为网络计划的优化、调整和执行提供明确的时间参数。同时，这个工程网络图可以作为进一步计划的基础，因为依据存在的依赖关系，能够给出多种不同的计划方案。本例可以给出的一个分析结果是，该项目最小的可能的持续时间为 3+2+3+5=13，它是那些机动时间为零的任务的持续时间之和。我们把能够确定的最小持续时间的路径称为关键路径，每个处于关键路径上的工作包，如果遇到计划外的延期，那么意味着整个项目的拖延。因此，复杂项目计划中很重要的一项优化就是在关键路径上尽可能少的安排工作包，这样也就在计划中为尽可能多的工作包产生机动时间提供了可能。

另外，项目计划的最终实施还直接依赖于人力等资源，若某些员工只在固定的时间可用，则计划也需要遵循这些先决条件进行调整。总之，项目计划及所有资源的总体分配方案是彼此依赖的复杂系统，不仅要使员工的工作负担尽量均衡，也要使项目尽可能顺利地开展下去，这对项目计划提出了更高的要求。

12.2.5　项目组织与甘特图

项目计划要能够方便、直观地将相关信息确定并展示出来，包括项目中各任务的时间、人力等资源安排，而且要明确各任务的主要责任人。项目计划的制订者，要熟悉项目组中每位成员的能力和擅长的技术，并按照岗位的技术和能力要求，科学、合理地分配每位人员的工作任务。

图 12.14 概括说明了几种项目计划的影响因素，项目计划的制订需要在各种因素的作用下渐进地迭代进行，不可以生搬硬套、完全主观地依赖一成不变的套路，必须要能够根据项目计划阶段中不同的限制条件和实际状况对计划进行适时、动态的调整。

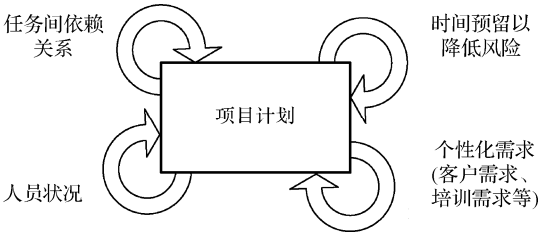


图 12.14　影响项目计划的主要因素

项目计划经常使用一种称为甘特图(Gantt Chart)的图形来对计划进行清晰的描述，包括各工作任务的完成进度、人员分配等详细信息。图 12.15 为甘特图的一个例子，其中除了工作包，还使用了一种实心菱形的符号，给出里程碑的位置，表示在对应的时间点可以对现有进度进行评审，并根据评估结果对计划进行较大调整，甚至终止项目。这里有两种里程碑：一种为内部里程碑，主要是在开发团队内部进行的进度评审；另一种是外部里程碑，客户需要在此了解当前项目的进展并对产品进行部分验收。外部里程碑的时间点通常会在项目早期达成共识并纳入合同，并在计划中作为约束条件进行考虑。内部里程碑的设定则可以稍微灵活，并可根据前面的里程碑的结果对后面的里程碑的计划适当调整。

ID	任务名称	开始时间	完成	持续时间
1	需求确认	2014/3/17	2014/3/31	11天
2	文献查阅、实地调研	2014/4/1	2014/7/30	87天
3	数据库设计与数据整理	2014/14/5	2014/6/16	45天
4	模块划分与概要设计	2014/6/17	2014/7/18	24天
5	各模块详细设计与开发	2014/7/21	2014/11/10	81天
6	性能评估	2014/7/21	2014/8/19	22天
7	运行期实时风险评估	2014/8/22	2014/9/19	21天
8	大坝三维模型	2014/9/22	2014/10/22	23天
9	大坝时间序列模拟	2014/10/23	2014/11/20	13天
10	系统集成与测试	2014/11/11	2014/12/19	29天

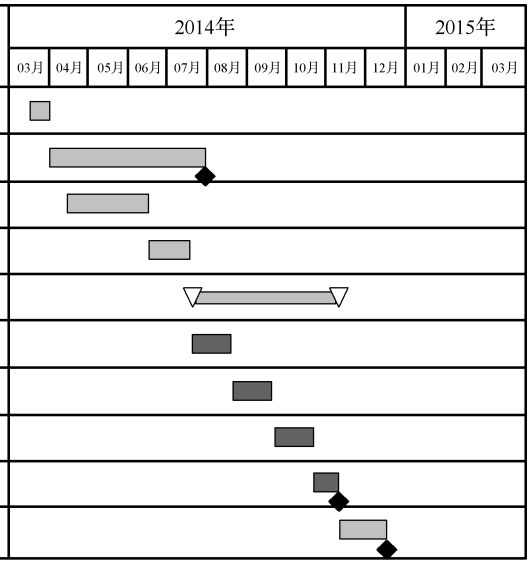


图 12.15　甘特图

12.3　项目计划跟踪控制

12.3.1　项目计划跟踪

项目经理的一个重要任务是随时掌握项目当前的进行状态，识别出潜在的风险或延期的征兆，并快速应对。为此，需要重点考虑以下两方面的信息。

(1)已进行的工作量(资源使用情况)。通常以“人小时”为单位进行度量，一般由专门的工作人员来记录和统计。

(2)任务完成的进度。通常使用百分数表示，需要开发者根据情况进行估算，以日报或周报等方式上报汇总。

在进行完成进度的跟踪时，可以将一个近似线性的模型作为参考，因为根据经验，任务完成的趋势从开始到 80%左右的速度是比较快的，然后会有较多的缺陷修正等维护任务，而使进度增长趋于缓慢。

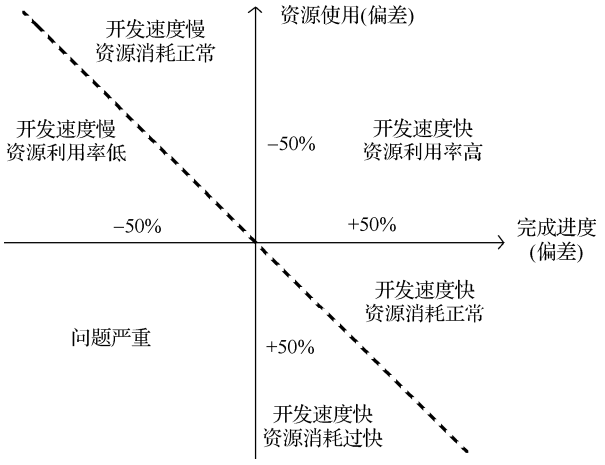


图 12.16　计划偏差的分析

项目经理通常将这两方面的数据与实际计划的偏离情况进行对比分析，图 12.16 给出了一个坐标系统，记录了项目整体或单个任务的进度跟踪情况。参照标准是对应的曲线与图中的虚线尽量接近。尽管曲线的位置处于虚线以上的上三角位置代表较高的开发效率和经济效益，但这样的偏离也要引起注意，因为处于右上象限表示任务在较少的资源消耗下可以快速完成，原因可能是项目计划时过高的不合理估算，项目经理要及时调整计划或者合理优化安排那些空闲的资源。

项目进行中的所有决定都需要在项目日志中进行记录，而且应由项目经理在项目结束后给出一个最终的项目总结评价，包括与客户交流的质量和效率、项目组人员的技术水平、开发工具的使用效率及使用的软件过程表现等方面内容。项目历史的记录和完善，对一个学习型和持续改进型团队来说，是宝贵的资源。

12.3.2　挣值分析

挣值分析(Earned Value)是对项目实施的进度、成本状态进行绩效评估的有效方法，是计算实际花在一个项目上的工作量成本，及该项目预计所需成本和完成该项目的预计日期的一种方法。

挣值分析是利用成本会计的概念评估项目进展情况的一种方法。传统的项目性能统计是将实际的项目数据与计划数据进行比较，计算差值，判断差值的情况。但是，实际的执行情况可能不是这样简单的，如果实际完成的任务量超过了计划的任务量，那么实际的花费可能会大于计划的成本。所以，应该计算实际完成结果的价值，这样就引出了挣值的概念，即到目前为止项目实际完成的价值。有了挣值，就可以避免只用实际数据与计划数据进行简单的减法而产生的不一致性。挣值分析模型如图 12.17 所示。

其中的输入参数定义如下。

(1)BCWS(Budgeted Cost of Work Scheduled，计划完成工作的预算成本)：到目前为止的总预算成本。它表示“到目前为止的计划成本是多少”或者说“到目前为止本应该完成的工作是多少”，它是根据项目计划计算出来的。

(2)ACWP(Actual Cost of Work Performed，已完成工作的实际成本)：到目前为止所完成工作的实际成本，它说明了“到目前为止实际的成本”，可以由项目组统计出来。

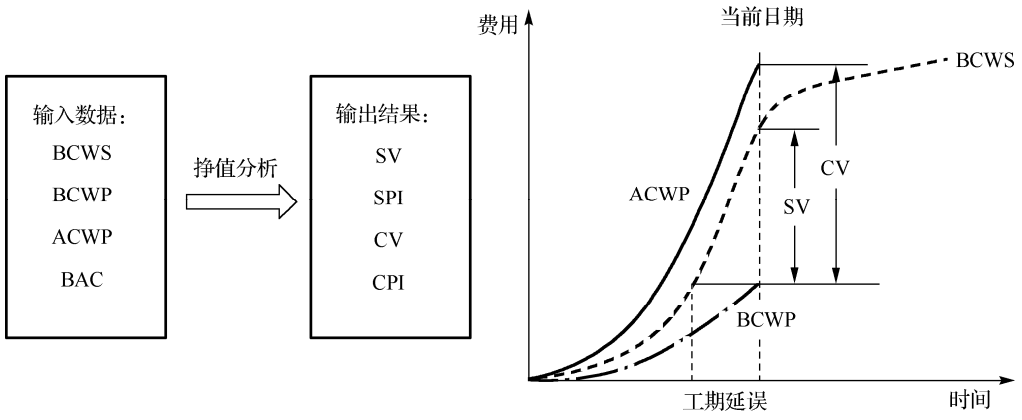


图 12.17　挣值分析模型

(3) BCWP (Budgeted Cost of Work Performed，已完成工作的预算成本)：又称挣值，到目前为止已经完成的工作的计划成本，它表示了“到目前为止完成了多少工作”。

(4) BAC (Budgeted At Completion，工作完成的预算成本)：项目计划中的成本估算结果，项目完成的预算总成本。

如果项目一切正常，ACWP、BCWP、BCWS 应该重合或接近重合。有了这些采集的数据信息，可以使用挣值分析模型来分析输出结果，输出指标的计算如下。

(1) 进度偏差 (Schedule Variance，SV) = BCWP–BCWS，若此值为零，表示按照进度进行；若为负值，则表示项目进度落后；若为正值，则表示进度超前。

(2) 成本偏差 (Cost Variance，CV) = BCWP–ACWP，若此值为零，则表示按照预算成本进行；若为负值，则表示项目超出预算成本；若为正值，则表示低于预算成本。

(3) 进度执行指标 (Schedule Performance Index，SPI) = BCWP/BCWS，项目挣值与计划值之比。若 SPI>1，则表示进度超前；若 SPI=1，则表示实际进度与计划进度相同；若 SPI<1，则表示进度延误。

(4) 成本执行指标 (Cost Performance Index，CPI) = BCWP/ACWP，项目挣值与实际费用之比。若 CPI>1，则表示低于预算，即实际费用低于预算费用；若 CPI=1，则表示实际费用与预算费用吻合；若 CPI<1，则表示超出预算，即实际费用高于预算费用。

在项目进行过程中，期望 SPI 和 CPI 趋近于 1，允许有一定的偏差，但是若偏差太大，则说明项目进度和成本控制出现了较大问题，需要采取措施调整项目计划，使得未来的 SPI 和 CPI 始终趋近于 1。

在使用挣值分析时，对于 BCWS、BCWP、ACWP 这些基本参数值，可以按照一定的时间段来统计计算，例如每周、每个月等。其中，挣值 BCWP 的计算是不容易的，需要考虑的原则有 50/50 规则和 0/100 规则，或者其他的经验加权法等。50/50 规则是指当一项工作任务已经开始，但是没有完成时，假定已经实现 50%的价值，当这个工作任务全部完成的时候，才实现全部的价值。0/100 规则是指当一项工作任务开始，但是没有完成时，不产生任何价值，即价值是 0，直到完成时才实现全部的价值。

举例说明，有 4 个工作任务，T1 全部完成，T2 完成一半，T3 刚开始做，T4 还没有开始做，到目前为止的各个指标如表 12.7 所示。

表 12.7　工作任务的 BCWS 和 ACWP 的取值

工作任务	BCWS（元）	ACWP（元）	任务总预算（元）
T1	2000	1900	2000
T2	900	750	1600
T3	200	300	1000
T4	0	0	1200

根据 50/50 和 0/100 规则可以分别计算出 BCWP，如表 12.8 所示。

表 12.8　工作任务的 BCWP 计算取值

工作任务	BCWS（元）	ACWP（元）	BCWP（元）（50/50 规则）	BCWP（元）（0/100 规则）
T1	2000	1900	2000	2000
T2	900	750	800	0
T3	200	300	500	0
T4	0	0	0	0
合计	3100	2950	3300	2000

因此，按照 50/50 规则计算出的 SPI=3300/3100=1.06，CPI=3300/2950=1.12，可以看出二者都比 1 大，表示项目进度超前，成本花费低于预算，是一个比较好的项目执行。注意，如果 SPI 和 CPI 比 1 大得过多，说明项目计划的预算不合理，过高地估计了一些任务计划预算，需要调整。

按照 0/100 规则计算出的 SPI=2000/3100=0.68，CPI=2000/2950=0.69，可以看出二者都比 1 小，表示项目进度滞后，成本花费高于预算，项目的执行需要调整。

从这个例子可以看出，使用不同规则计算输出的结果是不同的。

12.3.3　项目偏差控制

对项目进行了预算后，项目的执行过程可以按照预算内完成、低于预算完成或者超预算完成。超预算完成会给项目带来经济损失，所以控制项目执行预算是一个非常重要的管理过程，需要设定一个允许偏差范围，对超出允许偏差范围的挣值分析结果，需要采取措施纠正偏差。

通过进度偏差 SV、进度执行指标 SPI，可以分析进度问题，如果滞后严重，分析开发者资源使用情况，对正在做和未开始做的任务计划做出必要的调整，降低未来的偏差。

通过成本偏差 CV、成本执行指标 CPI，可以分析成本问题，如果成本超出当前预算，分析工作任务复杂性及其预算的合理性，对正在做和未开始做的任务计划做出必要的调整，降低未来的偏差。

对项目的进度和成本进行监控的同时，也应该对项目的质量、风险、人员等方面进行监控，只有它们的指标在计划的控制范围之内，项目的进度和成本控制才有意义。

12.4 软件质量保证

12.4.1 质量管理

时间、成本与质量在项目管理中常常一同提起。那么如何在时间、成本、质量这三个方面找到均合适的模式，并遵守这种模式，持续地进行管理工作呢？这就是质量管理的最终目标。

质量管理的学派和观点有很多，具有代表性有戴明理论、朱兰理论、克鲁斯比理论、田口玄一理论等。其中，戴明理论的核心是目标不变、持续改善和知识积累，预防胜于检验。朱兰理论的核心思想是“适用性”，适用性是通过遵守技术规范，使项目符合或者超过项目涉众的期望。克鲁斯比理论的核心思想是质量定义符合预先的要求，质量源于预防，质量的执行标准是零缺陷，质量是用非一致成本来衡量的。田口玄一核心思想是应用统计技术进行质量管理，通过损失函数来决定未满足目标产品的成本。

全面质量管理(Total Quality Management，TQM)是指通过全体员工的参与、改进流程、产品、服务和公司文化，达到在百分之百时间内生产百分之百的合格产品，以便满足顾客需求。TQM 是一种思想观念，一套方法、手段和技巧。

软件项目的质量管理是指，保证项目满足其目标要求所需要的过程，质量管理的关键是预防重于检查，事前计划好质量，而不是事后检查。任何软件开发项目中，质量不仅拥有发言权，而且对项目的成败拥有表决权甚至最终的否决权。质量不仅会对软件开发项目本身的成败产生影响，而且会对软件企业的形象、信誉、品牌带来影响。质量一般通过交付物标准来明确定义，这些标准包括各种特性及这些特性需要满足的要求。另外，质量还包含对项目过程的要求，例如规定执行过程应该遵循的流程、规范和标准，并要求提供过程被有效执行的证据。因此，质量管理主要就是监控项目的交付物和执行过程，以确保它们符合相关标准，同时确保不合格的内容能够按照正确方法排除。质量管理还可能对项目的客户应对质量做出规定，包括应对客户的态度、速度及方法。高质量来自满足客户需求的质量计划、质量保证、质量控制和质量改善活动，来自保证质量、捍卫质量和创造质量的卓越理念、规则、机制和方法。

质量管理过程的目的是确保项目满足需要执行的过程。主要过程是软件质量规划(Software Quality Planning)、软件质量保证(Software Quality Assurance，SQA)和软件质量控制(Software Quality Control)。软件质量规划确定与项目相关的质量标准及如何满足这些标准。软件质量保证通过定期的评估项目的整体性能，确保项目满足相关的质量标准。软件质量控制通过控制特定项目的状态，保证项目完全按照质量标准完成，同时确定质量改进的方法。

1. 软件质量规划

软件质量规划过程是确定项目应达到的质量标准，决定如何满足质量标准的计划安排和方法。合适的质量标准是质量规划的关键。只有做出精准的质量规划，才能指导项目的实施，做好质量管理。

质量规划主要指依据公司的质量方针、产品描述及质量标准和规则等制订出来实施策略，其内容全面反映用户的要求，为质量小组成员有效工作提供了指南，为项目小组成员及项目

相关人员了解在项目进行中如何实施质量保证和控制提供依据，为确保项目质量得到保障提供坚实的基础。

2. 软件质量保证

软件质量保证是为了提供信用，证明项目将会达到有关质量标准，而开展的有计划、有组织的工作活动。它是贯穿整个项目生命周期的系统性活动，经常性地针对整个项目质量计划的执行情况，进行评估、检查与改进等工作，向管理者、客户或其他方提供信任，确保项目质量与计划保持一致。

软件质量保证的目的是验证项目在软件开发过程中是否遵循了合适的过程和标准，是保证软件透明开发的主要环节。它贯穿了整个项目的始终。

软件质量保证的职责是确保过程的有效执行，监督项目按照指定过程进行项目活动；同时审计软件开发过程中的产品是否按照标准开发。软件质量保证的主要方法是质量审计，即产品审计和过程审计。为此，质量保证人员要定期对项目质量计划执行情况进行评估、审核与改进等工作，在项目出现偏差的时候提醒项目管理人员，提供项目和产品可视化的管理报告，通过各种手段来保证得到高质量结果的工作。

3. 软件质量控制

软件质量控制是确定项目结果与质量标准是否相符，同时消除不符的原因和方法，控制产品的质量，及时纠正缺陷的过程。软件质量控制是对阶段性的成果进行检测、验证，为质量保证提供参考依据。

软件质量控制主要用于发现和消除软件产品的缺陷。对于高质量的软件来说，最终产品应该尽可能达到零缺陷。而软件开发是一个以人为中心的活动，所以出现缺陷是不可避免的。因此，要想交付一个高质量的软件，消除缺陷的活动就变得很重要。消除缺陷是通过“评审”和“测试”这类质量控制活动实现的。软件质量控制方法有技术评审、走查、测试、返工、控制图、趋势分析、抽样统计、缺陷追踪等。

12.4.2 软件质量保证的内容

软件质量保证的内容主要包括：软件质量保证过程；具体的质量保证和质量控制任务(包括技术评审和多层次测试策略)；有效的软件工程实践(方法和工具)；对所有软件工作产品及其变更的控制；保证符合软件开发标准的规程；测量和报告机制。

软件质量保证涵盖了广泛的内容和活动，这些内容和活动侧重于软件质量管理，主要的要素如下。

(1)标准：IEEE、ISO 及其他标准化组织制订了一系列广泛的软件工程标准和相关文件。标准可能是软件工程组织自愿采用的，或者是客户及其他系统涉众责成采用的。软件质量保证的任务是要确保遵循所采用的标准，并保证所有的工作产品符合标准。

(2)评审和审核：技术评审是由软件工程师执行的质量控制活动，目的是发现错误。审核是一种由软件质量保证人员执行的评审，意图是确保软件工程工作遵循质量准则。例如，要对评审过程进行审核，确保以最有可能发现错误的方式进行评审。

除此之外，还包括测试、缺陷收集分析、变更管理、培训、供应商管理、风险管理等方面的活动。

12.4.3 软件质量保证的任务

软件质量保证团队的行动纲领是协助软件团队实现高质量的最终产品。CMMI（Capability Maturity Model Integration，软件能力成熟度模型集成）中推荐了一套质量保证活动，包括质量保证计划、监督、记录、分析和报告，这些活动由独立的软件质量保证团队执行和完成，他们的任务如下。

（1）编制项目质量保证计划。该计划作为项目计划的一部分，并且经所有系统波涉众评审。软件工程团队和软件质量保证团队进行的质量保证活动都受该计划支配。在计划中需要确定要进行的评估、要进行的审核和评审、适用于项目的标准、错误报告和跟踪的规程、软件质量保证组产出的工作产品及将提供给软件团队的反馈意见。

（2）参与编写项目的软件过程描述。软件工程团队选择完成工作的过程。软件质量保证团队审查该过程描述是否符合组织方针、内部软件标准、外部要求的标准（例如 CMMI），是否与软件项目计划的其他部分一致。

（3）评审软件工程活动，以验证其是否符合规定。软件质量保证团队识别、记录和跟踪偏离过程的活动，并验证是否已做出更正。

（4）审核指定的软件工程产品，以验证是否遵守规定。软件质量保证团队审查选定的产品，识别、记录并跟踪偏差，验证已经做出的更正，并定期向项目经理报告其工作成果。

（5）确保根据文档化的规程，记录和处理软件工程和工程产品中的偏差，包括在项目计划、过程描述、适用的标准或软件工程工作产品中存在的偏差。

（6）记录各种不符合项并报告给上层管理人员。跟踪不符合项，直到解决。

除了这些任务，软件质量保证团队还协调进行软件配置变更的控制和管理，并帮助收集和分析软件度量数据。

12.4.4 软件质量保证计划

软件质量保证计划，也称 SQA 计划，为软件质量保证提供了一张路线图，该计划由软件质量保证团队来制订，作为各个软件项目中软件质量保证活动的模板。

SQA 计划应包括：计划的目的和范围；软件质量保证覆盖的所有软件工程产品的描述，如模型、文档、源代码等；应用于软件过程中的所有适用的标准和习惯做法；软件质量保证活动和任务（包括评审和审核）及它们在整个软件过程中的位置；支持软件质量保证活动和任务的工具和方法；软件配置管理的规程；收集、保护和维护所有软件质量保证相关记录的方法；与产品质量相关的组织角色和责任。

12.5 风险管理

软件风险是指软件开发过程中及软件产品本身可能造成的伤害或损失。风险关注未来的事情，这意味着，风险涉及选择本身包含的不确定性。软件开发过程及软件产品都要面临各种决策的选择。当在软件工程领域考虑风险时，我们要关注以下问题。

（1）什么样的风险会导致软件项目的彻底失败？

(2)用户需求、开发技术、目标计算机及所有其他与项目有关的因素的改变将会对按时交付和总体成功产生什么影响？

(3)采用什么方法和工具，需要多少人员参与工作的问题，如何选择和决策？

(4)软件质量要达到什么程度，才是“足够的”？

当没有办法消除风险，甚至连试图降低风险也存在疑问时，这些风险就是真正的风险了。在我们能够标识出软件项目中的真正风险之前，识别出所有对管理者和开发者而言均明显的风险很重要。

被动风险策略是针对可能发生的风险来监督项目，直到它们变成真正的问题时，才拨出资源处理它们。更普遍的是，开发者对风险不闻不问，直到发生了错误，才采取行动，试图迅速地纠正错误。这种管理模式常常被称为“救火模式”。当补救的努力失败后，项目就处在真正的危机之中了。

风险管理的一个更好的策略是主动策略。主动策略在技术工作开始之前就已经启动了——标识出潜在的风险、评估它们出现的概率及产生的影响，对风险按重要性进行排序。然后建立一个计划来管理风险。主动策略的主要目标是预防风险。但是，不是所有的风险都能够预防，所以必须建立一个应对意外事件的计划，使其在必要时能够以可控、有效的方式做出反应。

项目的风险管理内容如图 12.18 所示，其中描述了主要的风险活动及其关系，包括风险识别、风险分析、措施计划、措施执行、结果评估、优化、风险数据库几个主要部分。

1．风险识别

风险识别步骤一般在项目开始时就已经进行了，用来识别出项目开发过程中潜在的风险。虽然风险不能立即被全部识别出来，但越早发现风险，后续的步骤就越容易。

2．风险分析

对于每个风险来说，风险分析需要确定两个影响因素：一是该风险出现的概率有多大；二是一旦出现风险，其破坏程度如何。可以将这两项值的乘积作为该风险的优先级进行管理，评估可能的后果，并考虑相应的应对措施。

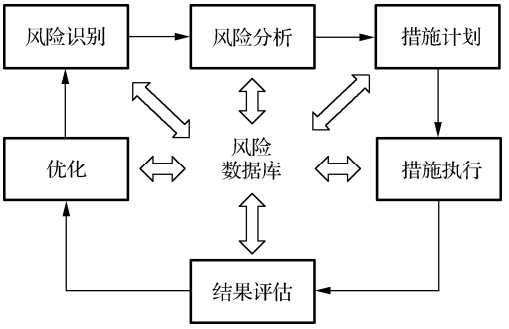


图 12.18　风险管理过程

3．措施计划

措施计划对每个威胁的风险制订最小化其危害的应对计划，如原型的开发、员工的培训、通过协议对风险的转移等，制订每项措施的成功指数。

4．措施执行

措施执行是指按照措施计划执行应对措施。

5．结果评估

结果评估在措施执行后规定的时间点(如对项目产品评审时)，依据每项措施的评估指数验证该措施是否成功。

6．优化

优化是指对某项风险及其应对措施的优化，包括对措施的调整或者指定更有效的措施。优化也指对风险管理过程本身的优化，因为风险管理过程也同样需要持续改进和完善。

7．风险数据库

风险数据库是指组织级上建立的经验数据库，将不同的项目及它们的风险管理信息在数据库中进行存储，为现有项目风险的评估提供信息参考，同时使得未来项目的经验在组织级持久化。风险分析的结果数据样例如表 12.9 所示。

表 12.9　风险分析的结果数据样例

风险名称	类别	发生概率	影响程度	风险排序
用户变更需求	产品规模	80%	5	1
规模估算不准确	产品规模	70%	5	2
开发者变动	人员数目及其经验	60%	4	3
客户对计划有异议	商业影响	50%	4	4
产品交付期限提前	商业影响	50%	3	5
软件负载超出计划	产品规模	30%	4	6
技术水平达不到	技术情况	20%	2	7
缺少对开发工具的学习	开发环境	40%	1	8
成员缺乏开发经验	人员数目及其经验	10%	3	9

12.6　项目人员构成与沟通

项目成功的另外一个重要因素就是社交环境，这也是纯开发者不擅长的领域。正如在敏捷软件过程模型中强调的，个体和互动胜过过程和工具，因此很难想象没有互动的团队的软件开发是什么样的。开发者通过项目构成了一个小的社会环境，团队成员在一起高效地工作，促成最终项目的成功，并享受在一起工作的愉快和成就感。社交技能的培养在教育或培训中也越来越受到重视，因为有效的沟通能够激发每位员工的潜能。

12.6.1　项目人员构成

尽管过程的思想在成功项目的开发中起到重要的作用，但在过程的层面上也不能解决所有问题。图 12.19 给出了项目的整体视图，并将项目及其成功的因素划分为过程层面、能力、社交层面、企业环境 4 个部分。

(1) 过程层面：包括企业中所有的过程定义及其描述。

(2) 能力：包括所有工作人员能够掌握和应用的所有技术和构件。

(3) 社交层面：包括工作人员之间所有的交互情况、交互特点及交互质量。

(4) 工作环境：包括所有组织级的能够影响项目成败的制度和设施。

项目的整体视图指出，一个领域的问题是不会在另外的领域中被解决的。如果两个工作

人员间不能相互忍受并不断争吵，只能尽可能地界定他们的职责范围，而对于发生冲突的情况，冲突的解决或缓和只能在社交层面上进行。该视图也指明不能通过过程模型中清晰的过程定义来弥补能力的缺失，因为只有先具有能力(Know How)，才会使过程变得可用。

项目团队刚开始组建时，往往集合了很多互不相识的工作人员，他们以前可能没有共事过。这样的团队要有一个熟悉和信任的过程，这个过程需要时间，项目经理也需要适应这个过程。



图 12.19　项目的整体视图

(1)组建(Forming)：项目成员之间相互熟悉，彼此了解。团队对工作专注不够，效率低下。

(2)风暴(Storming)：团队花费大量的时间相互融合，包括职责的确定和工作的分配。团队不稳定，效率低下。

(3)规范(Norming)：团队形成规范，每个人开始关注怎样工作能最好地达到目标。团队工作效率得到提升。

(4)行动(Performing)：团队处于集中工作的稳定状态，每个成员知道自己的职责并了解与其他成员如何配合。团队的工作效率极高。

在社交中扮演的角色与人员的类型是密切关联的。对于人员的角色确定，需要有特别的考虑。一种建设性的方法是将项目人员的类型按照特点划分为 9 类，每类都含有积极和消极的一面。不同类型的人员可适用于不同的项目，并且对于项目而言，得益于若干不同人员类型的组合使用。对于每种人员类型的特点，并不是每种人员只能安排一种角色与之对应，而是要发现哪些角色对某种人员的类型更偏重，哪些次之。这 9 类人员可描述如下。

(1)专家(Specialist)：专注于新技术的研发并对自己的能力感到骄傲，不太关注其他人。

(2)润饰者、完工者(Completer、Finisher)：注重细节，做事情不会半途而废，工作积极主动；做事总是亲力亲为，通才但不擅长外交。

(3)实现者(Implementer)：务实者，具备处理系统复杂任务的能力，忠实、性格内向、直率；缺乏创新。

(4)团队者(Team Worker)：平衡、外向、不拔尖、善于倾听，与大家在团队中自由地通信，能够迅速适应新的形势；回避针锋相对。

(5)监督评估者(Monitor Evaluator)：平衡，对当前形势能够严肃、清晰地看待，高智商、高创造力，能够快速发现错误；有些悲观，其敏锐的批判能力降低了对新事物的接收能力。

(6)领导者(Shaper)(开发任务)：积极进取，愿意领导和协调团队，找出问题的解决方法，外向、有责任心；容易冲动、急躁，有时过于追求速度。

(7)协调者(Coordinator)(项目社交)：稳定，以我为中心，外向、纪律性强，团队的领导者，为外部的目的和目标努力，善于沟通，项目角色安排合适；缺少创新，在项目推进中过于安静。

(8)资源投资者(Resource Investigator)：休闲、沟通型，善于从外部吸取灵感，善于外交，善于经商，具有丰富的社会关系；依赖外部的机遇(动机)。

(9)电厂(Plant)：能够提出原创的想法、动机和建议，以我为中心，高智商，聚焦主要工作；内向，有干扰和攻击性，注重宏观，较难接收外界的批判者质疑。

每位读者在其中一定能够找到一个或多个自己适合的类型，如果是计算机方面的科学家可能更喜欢专家和电厂的角色。对不同角色类别的了解，有助于认识到项目开发对人员的多样性需求，他们在业务、社交、管理等能力上能够互补，发挥出 1+1>2 的作用，他们的团队合作能够引领项目走向成功。

12.6.2 项目人员沟通

在不同类型持不同见解的人员之间建立沟通的渠道和氛围，可利用四方沟通模型。四方沟通模型，又称通信方(Communication Square)或四耳朵模型(Four-ears)，是由 Friedemann Schulz von Thun 提出的一种沟通模型。模型认为，每条消息都由四个方面构成，但每个方面强调的重点有所不同。这四个方面分别为：事实(Fact)、自我揭示(Self-Revealing)、关系(Relationship)和诉求(Appeal)，模型如图 12.20 所示。例如，项目经理对某个开发者说：“你应对这个类进行注释”，以这个消息为例按照模型的四个方面进行介绍，其中无论说话的表情，还是重音的位置，都是话语含义的重要成分。

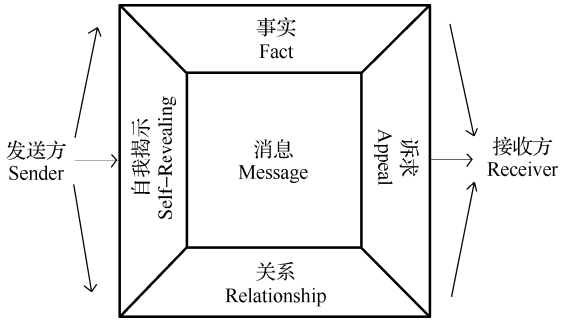


图 12.20　四方沟通模型

一般来说，话语的信息主要来自其组成部分，通过消息的事实和关系两个方面表达。这个例子中的事实传达了“负责开发这个类的人还没有对它进行注释”的事实。

自我揭示表示的含义是“我揭示自己什么”，消息中包含了发送方有意识的和无意识的自我倾向，与强调的重点或手势有很大的关系。示例中，项目经理可能要表明一下他的疑虑，就是他不确定是否这个项目的要求得到了满足。对于接收方，在自我揭示这个层面上可能会对原消息有不同的理解，他可能认为项目经理现在对项目目前的情况不是非常满意。因此，接收方接收消息后总是在寻求隐含在消息事实后的内容是什么。

消息中还蕴含着发送方和接收方的关系信息，即对话的双方对待彼此的态度，应如何相处。例如，项目经理可能在想那个开发者还是有潜力的，如果他对他的工作再严格一点，会有好结果。而那个开发者可能在关系层面上想到项目经理认为自己不具备解决最简单问题的能力。这通常是交流中最具敏感的层面，如果对话双方彼此误解，就会在此层面上升级。

最后一个层面就是诉求层，表达“我想从你那里获得什么”。示例中，项目经理想要的效果是开发者应系统化地工作。同时，接收方也存在另外一种解释，如项目经理希望他应首先做好最基础的任务。

无论是从开发的角度，还是从管理角度，危机管理是一项重要的沟通层面的内容。危机管理的一部分工作就是要能够识别出恶性的沟通循环。这种循环会通过强硬的言语逐渐加深

参与者之间的负面印象。图 12.21 给出了对此过程的描述，暂且不管该循环是如何开始的，员工首先感受到了来自领导层的巨大压力，但是他们害怕失去工作，在工作压力和问题下继续选择努力工作。领导者往往认为他们的决定正在发挥积极的作用，当偶尔有抱怨和挑剔产生时，他们将作为例外进行处理。这将导致他们的领导方式没有改变，员工依然承受压力。这样的恶性循环会一直持续，并在最后某个时刻爆发并终止，如某个大型项目的失败或者员工辞职。

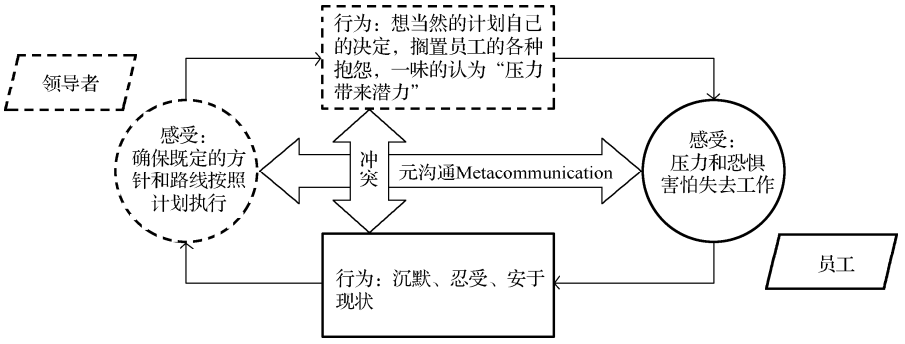


图 12.21　恶性的沟通循环

对这种循环模型的了解能够指导我们多从交流的“元沟通”(即感受)，而不是“行动层”进行改善，多关怀员工的实际感受强于生硬的命令。如在上述场景下，可通过员工代表(工会)给予员工更多的鼓励，以此形成一个建设性的、开放的对话环境。一方面员工具有自由表达思想的权力，另一方面员工产生对公司的信任和依赖的情感，这样在元沟通上的交流就得以顺利进行。

12.7　习题

1．根据版本管理的机制设计一个版本仓库，并使用工具实现它。

2．选择使用第 3 章习题中描述的一个系统需求，完成系统规模、工作量和成本估算的过程并给出结果。

3．给定如图 12.22 所示的工程网络图，补充计算每个工作包的计划安排，并指出其关键路径。

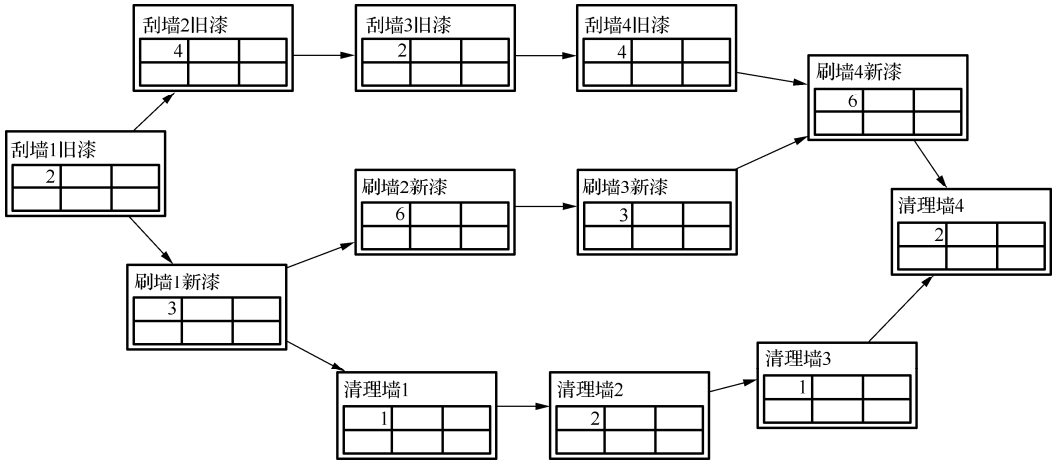


图 12.22　工程网络图

4．在上题的基础上，为 2 人的团队使用甘特图制订计划安排，要求指定每项任务的责任人，并标出可能的里程碑位置。

5．根据表 12.10 中项目的成本数据，计算出项目的 BAC，然后根据 50/50 和 0/100 规则，计算 BCWS、ACWP、BCWP，并进一步计算 SPI 和 CPI，说明在 7 月 1 日时项目的状况如何。

表 12.10　项目的成本数据

工作任务	估算的成本(千元)	到目前为止实际成本(千元)	估计完成日期	实际完成日期
T1	5	10	1 月 25 日	2 月 1 日
T2	20	15	4 月 1 日	3 月 15 日
T3	80	60	5 月 15 日	4 月 1 日
T4	30	40	6 月 15 日	5 月 20 日
T5	40	45	7 月 1 日	7 月 1 日
T6	50	—	9 月 1 日	—

6．软件质量保证的工作任务有哪些？

第 13 章　软件过程管理与改进

软件项目级管理使得软件项目开发过程的可控化和可视化得到了良好的实现，软件质量也得到了初步的保证。对于从事多个项目开发的企业组织，不但需要对每个项目有各自的管理过程，更需要建立一个统一、明确定义的组织级软件过程，使得所有的项目顺利执行，同时在执行中不断地改进和完善组织级过程体系。

我们意识到软件开发的产品与服务变得日益复杂，而复杂产品或服务的开发可能在内部而另一些组件则对外采购，然后集成起来，成为最终的产品或服务。对如此复杂的开发与维护过程，组织必定要有能力进行管理并控制，所以，有效的组织级管理是成功的关键。CMMI 是企业级的软件开发过程模型和质量标准，规范的软件开发是从项目级管理开始的，这是 CMMI 中描述和要求的，我们需要掌握基本项目管理的知识和方法。CMMI 更重要的是给我们提供了一个组织级的管理模型、标准、改进框架。为了系统地解决软件质量和过程管理问题，卡内基梅隆大学软件工程研究所经过多年努力，先后提出了能力成熟度模型集成(CMMI)、个体软件过程(PSP)和团队软件过程(TSP)，使软件过程框架形成一个包含 CMMI、PSP 和 TSP 三者的严密的整体，本章重点阐述以此为基础的组织级的软件过程体系。

一般认为，CMMI/TSP/PSP 是目前世界上最好的软件过程管理的模式之一，CMMI 提供了框架和目标，PSP 针对个人进行优化，TSP 针对团队进行优化。本章将对相关知识进行阐述，并深入说明。

13.1　软件过程管理

13.1.1　过程改进

很多与开发相关的领域越来越重视软件过程的概念，这已经是一个不争的事实。但为什么从 20 世纪 90 年代中期开始，软件过程的概念和思想越来越受到人们的重视，如何确定和组织高质量的软件开发过程？

质量的历史经历了产品发展历史的若干个阶段，其基本目标就是提供给客户满意的产品。质量的概念始于产品向客户的最终交付，最终交付意味着客户对进行产品检验并接收的过程，又称为质量控制。到了 20 世纪初，人们逐渐不满足于简单的质量控制活动，因为质量控制总是发生在产品成型之后，是一个事后的被动过程。通过对产品的最终控制，可以进行残次品数据的收集并进行分析，找出解决的措施，进而减少不合格产品的数量，这是一个主动应对的过程。对不合格产品的清点是最常用的对过程的质量度量。另外一个重要的质量措施是引入生产控制，以便能够在早期发现缺陷产品并识别出产品生产过程中存在的问题。这种方法多用在质量保证措施的改进上。

另外一个很自然的问题就是要弄清楚为什么某个生产过程的缺陷产品数量少于其他过程，从对过程的持续改进的角度来说，这也是质量管理的起点。这种质量的观点要求，不仅要重视产品本身的质量，而且应该注重产品生产过程的质量，因为在好的过程下会以极高的概率产生高质量的产品。产品的生产过程不是一蹴而就的，而是需要不断经过改进而提高的，发现现有过程的问题并制订措施完善，是过程改进不变的宗旨。图 13.1 给出了互为补充的四个项目因素，它们的作用是联动的，如并不能简单地提高开发速度，而不增加预算或者降低质量(忽略控制)或者去除某些功能。但我们可以找到某种更好的开发过程，保证期间问题较少，因而可以实现快速的开发，并产生高质量的产品。

承认并坚持过程改进带来的潜在积极作用，是质量管理的核心任务。要善于站在巨人的肩膀上，在别人的工作基础上进行过程的利用和改进。前面的章节已经就一些开发方法和它们的应用环境进行了说明，但由于项目领域的多样性及边界约束的复杂性，一个一成不变的通用开发过程是不存在的。好的开发过程的产生依赖很多具体的方面，如业务领域、公司长期目标等。另外，客户群体的业务领域和需求经常会发生改变，应通过过程实现快速响应和应对变化。这些都是质量管理的范畴，总之，过程的裁剪定制和灵活性是质量管理的目标和准则。

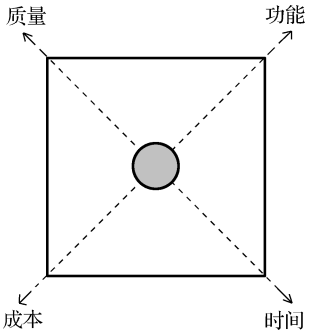


图 13.1　联动的项目因素

13.1.2　能力成熟度模型

能力成熟度模型(Capability Maturity Model，CMM)具有很多的变种，最后统一起来形成 CMMI。CMM 的初衷是为美国国防部(DoD)提供对软件开发承包商的资质评估，以增加项目开发成功的概率。此项目后来由卡内基梅隆大学的软件工程研究所负责，并形成了一套系统的评估标准，这也为软件开发组织提供了认证的标准和路线。模型中将过程的能力和成熟情况按照等级或成熟度进行了划分，如图 13.2 所示。

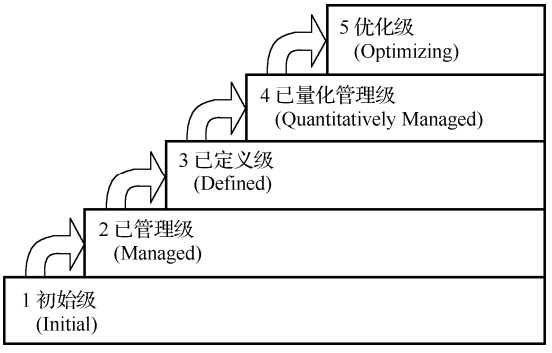


图 13.2　阶段式 CMMI 的级别

1. 初始级(Initial)

开发的初级阶段，没有引入任何系统化的过程控制。开发过程存在很大的随意性，开发结果存在较大的不确定性，并且难以理解和回顾，开发过程经常返工，工作量翻倍是常态。被动地等待问题的出现，采用救火式的处理方式，项目成败极大地依赖员工的技能和个人的承诺。

2. 已管理级(Managed)

这个阶段的项目可以再现，总结出了项目开发的特点和管理经验，项目管理起到了重要的作用。引入了关键的子过程：需求管理(REQM)、项目计划(PP)、项目跟踪和控制(PMC)、过程和产品质量保证(PPQA)、配置管理(CM)、供应商协议管理(SAM)和度量与分析(MA)。但仍然采用被动的问题应对方式。

3. 已定义级(Defined)

所有已有的过程都进行了统一的文档化，它们能够被理解和利用，并为其他项目提供一个统一的框架。对过程的分析综合和协调是管理的核心，为此需要以下子过程的支持：需求定义(RD)、验证(Val)、确认(Ver)、技术方案(TS)、风险管理(RSKM)、组织级过程聚焦(OPF)、组织级过程定义(OPD)、组织级培训(OT)、集成项目管理(IPM)、决策分析和决定(DAR)。采用主动的问题应对方式。

4. 已量化管理级(Quantitatively Managed)

为了识别出哪些过程改动会带来什么样的质量变化，需要对过程和产品质量进行量化的度量。每个过程的表现需要能够度量，并具有基于量化结果进行自我分析的能力，通过以下两个子过程进行支持：组织级过程性能(OPP)和量化项目管理(QPM)。

5. 优化级(Optimizing)

在此阶段，公司能够基于前面的基础阶段确定更合理的优化目标，及时识别并做出必要的过程调整。包含的子过程包括：组织级革新与实施(OID)、原因分析与解决(CAR)。

开发的能力与成熟度阶段性的改善过程和路线通过图 13.3 进行了可视化的描述。经过阶段 1 的杂乱无章的项目组织，在阶段 2 中的软件开发中融入了软件工程的思想，在软件过程的指导下进行开发，并在阶段 3 中对过程的各个环节的工作进行细化，然后在阶段 4 中对各个指标进行客观的量化，如评估出项目持续时间，最后在阶段 5 中根据新出现的问题或约束条件进行分析，并对过程进行持续改进。

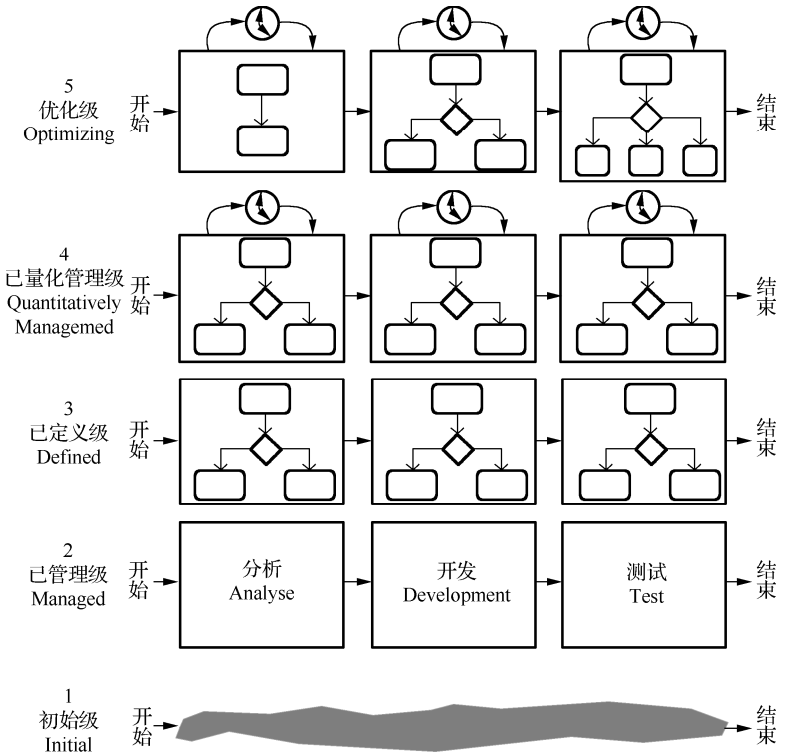


图 13.3　CMMI 与过程改进

CMMI 二级及以上的子过程又称过程域(Process Area)。每个过程域具有一系列的目标，包括特定目标(Specific Goals)和通用目标(General Goals)，并给出了它们对应的特定实践(Specific Practice)和通用实践(General Practice)。这里主要关注的是特定目标，它们是指那些受制于企业的某些具体行为的过程片段，也是企业进行过程改进的重点内容。图 13.4 给出了一个通过 CMMI 进行过程改进的简单示意，左侧是一个产品计划的原始步骤，其中的活动没有经过特别的优化，右侧参考了 CMMI，并对模型中的工作包定义进行了具体的改进。

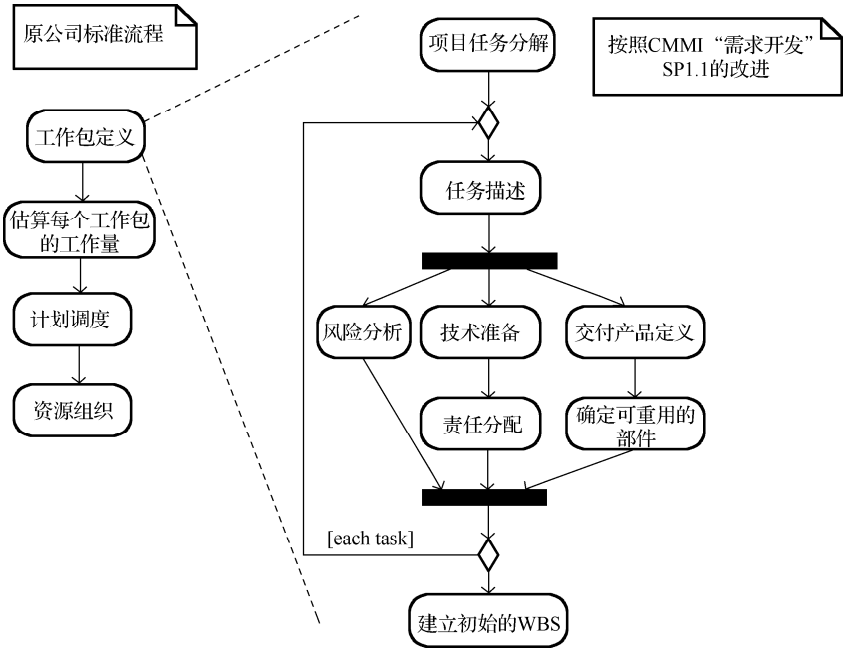


图 13.4　使用 CMMI 进行过程改进

实践表明，企业成熟度向高一级别的发展需要有足够的时间积累，一般至少为两年。软件过程不能纸上谈兵或者由上到下命令式推进，必须要得到所有员工的认可并作为必不可少的手段融入日常工作中，潜移默化地影响员工的做事方式，统一思想，行动一致，这就是企业文化的作用。

图 13.2 给出的实际上为 CMMI 阶段模型，每个阶段定义了若干个过程域，完成了过程域规定的要求，则可以过渡到下一个阶段。这类似在大学四年里，每年安排一定的课程和学分要求，当通过所有的考核时，可以升级到下一个年级。因此，CMMI 阶段模型旨在提高综合能力和成熟度。CMMI 还提供了一种连续模型供使用，连续模型强调的是对某些能力的重视和不断优化，而不是明显的阶段划分和引领，注重的不是综合能力和全面成熟度的提升，对连续模型本章不做展开介绍。

13.1.3　CMMI 过程域

过程域，简单地说就是做好一个事情的某一个方面。对应软件开发来说，就是做好软件开发的某一个方面。CMMI 中过程域(PA)的主要内容分为四大类 22 个，其中 2～3 级有 18 个，4～5 级有 4 个。

1. 过程管理类

(1) OPD (Organizational Process Definition)，组织级过程定义：建立和维护有用的组织过程资产。

(2) OPF (Organizational Process Focus)，组织级过程焦点：在理解现有过程强项和弱项的基础上计划和实施组织过程改善。

(3) OT (Organizational Training)，组织培训：增强组织各级人员的技能，使他们能有效地执行任务。

2. 项目管理类

(1) PP (Project Plan)，项目计划：保证在正确的时间有正确的资源可用。为每个人员分配任务、协调人员。根据实际情况调整项目。

(2) PMC (Project Monitoring and Control)，项目监督与控制：通过对项目的跟踪与监控活动，及时反映项目的进度、费用、风险、规模、关键计算机资源及工作量等情况，通过对跟踪结果的分析，依据跟踪与监控策略采取有效的行动，使项目能在既定的时间、费用、质量要求等情况下完成。

(3) SAM (Supplier Agreement Management)，供应商协议管理：旨在对以正式协定的形式从项目之外的供应商采办的产品和服务实施管理。

(4) IPM (Integrated Project Management)，集成项目管理：根据从组织标准过程剪裁而来的集成、定义的过程对项目和系统涉众的介入进行管理。

(5) RSKM (Risk Management)，风险管理：识别潜在的问题，以便策划应对风险的活动和在必要时在整个项目生命周期中实施这些活动，缓解不利的影响，实现目标。

(6) REQM (Requirement Management)，需求管理：需求管理的目的是在客户和软件项目之间就需要满足的需求建立和维护一致的约定。

3. 工程类

(1) RD (Requirement Development)，需求开发：需求开发的目的在于定义系统的边界和功能性、非功能性需求，以便系统涉众和项目团队对开发的内容达成一致。

(2) TS (Technical Solution)，技术解决方案：开发、设计和实现满足需求的解决方案。解决方案的设计和实现等都围绕产品、产品组件和与过程有关的产品。

(3) PI (Product Integration)，产品集成：从产品部件组装产品，确保集成产品功能正确并交付产品。

(4) VAL (Validation)，确认：确认产品或产品部件在实际应用下满足应用要求。

(5) VER (Verification)，验证：验证选定的工作产品满足需求规格。

4. 支持类

(1) CM (Configuration Management)，配置管理：建立和维护在项目的整个软件生命周期中软件项目产品的完整性。

(2) PPQA (Process and Product Quality Assurance)，过程和产品质量保证：为项目团队和管理层提供项目过程和相关工作产品的客观信息。

(3) MA (Measurement and Analysis)，测量与分析：开发和维持度量的能力，以便支持对管理信息的需要。

(4) DAR (Decision Analysis and Resolution)，决策分析与解决：应用正式的评估过程，依据指标评估候选方案，在此基础上进行决策。

第 4 级除第 2、3 级所涵盖的 18 个过程域外，增加了以下 2 项。

(1) OPP (Organizational Process Performance)，组织过程性能：建立与维护组织过程性能的量化标准，以便使用量化的方式管理项目。属于过程管理类。

(2) QPM (Quantitative Project Management)，量化的项目管理：量化管理项目已定义的项目过程，以达成项目既定的质量和过程性能目标。属于项目管理类。

第 5 级队以上 20 个过程域外，增加了以下 2 项。

(1) OPM (Organizational Performance and Management)，组织的绩效与管理：选择并推动渐进创新的组织过程和技术改善，改善应是可度量的，所选择及推动的改善需支持基于组织业务目的的质量及过程执行目标。属于过程管理类。

(2) CAR (Causal Analysis and Resolution)，因果分析与解决：识别缺失的原因并进行矫正，进一步防止缺失再次发生。属于支持类。

过程域对应的类别和能力成熟度等级如表 13.1 所示。

表 13.1　能力成熟度集成 (CMMI) 核心过程域

缩　写	英文名称	中文名称	领域类别	成熟度等级
CAR	Causal Analysis and Resolution	因果分析与解决	支持类	5
CM	Configuration Management	配置管理	支持类	2
DAR	Decision Analysis and Resolution	决策分析与解决	支持类	3
IPM	Integrated Project Management	集成项目管理	项目管理类	3
MA	Measurement and Analysis	测量与分析	支持类	2
OPD	Organizational Process Definition	组织级过程定义	过程管理类	3
OPF	Organizational Process Focus	组织级过程焦点	过程管理类	3
OPM	Organizational Performance Management	组织的绩效与管理	过程管理类	5
OPP	Organizational Process Performance	组织过程性能	过程管理类	4
OT	Organizational Training	组织培训	过程管理类	3
PMC	Project Monitoring and Control	项目监督与控制	项目管理类	2
PP	Project Planning	项目计划	项目管理类	2
PPQA	Process and Product Quality Assurance	过程和产品质量保证	支持类	2
QPM	Quantitative Project Management	量化的项目管理	项目管理类	4
REQM	Requirements Management	需求管理	项目管理类	2
RSKM	Risk Management	风险管理	项目管理类	3
SAM	Supplier Agreement Management	供应商协议管理	项目管理类	2
RD	Requirement Development	需求开发	工程类	3
TS	Technical Solution	技术解决方案	工程类	3
PI	Product Integration	产品集成	工程类	3
VAL	Validation	确认	工程类	3
VER	Verification	验证	工程类	3

过程域的描述使用了一个统一的框架结构，如图 13.5 所示。也就是说，每个过程域都需要有目的陈述、简介、相关过程域，然后要有特定目标及其达成目标的特定实践，特定实践中包含了一系列具体实施的子实践及使用的工作产品实例。还需要有通用目标，通用目标的达成依赖于实施通用实践，每个通用实践也包含一系列的子实践和对通用实践的详细说明。

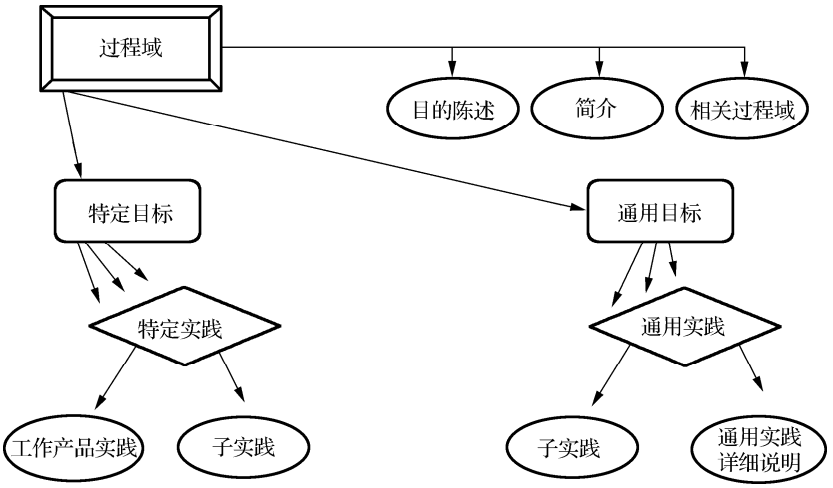


图 13.5　过程域的描述

它们的详细含义如下。

(1)目的陈述：描述了过程域的目的。例如，“组织级过程定义”过程域的目的是“组织级过程定义(Organizational Process Definition，OPD)的目的在于建立并维护一套可用的组织级过程资产、工作环境标准及团队规则与指南。”

(2)简介：过程域的简介描述了过程域所涉及的主要概念。例如，“项目监督与控制”过程域的简介是“当实际状态与预期情况显著偏离时，要酌情采取纠正措施。”

(3)相关过程域：相关过程域列出了至相关过程域的引用，反映过程域之间高层次的关系。例如，“项目计划”过程域的相关过程域中的一条引用是“参阅‘风险管理’过程域，以进一步了解如何识别、分析并缓解风险。”

(4)特定目标：特定目标描述了为满足过程域而必须呈现出的独特特征，在评估中用于帮助确定过程域是否得到满足。例如，“配置管理”过程域中的一个特定目标是“基线的完整性得到建立与维护。”只有特定目标陈述才是必需的模型组件。特定目标的标题(前面标有目标编号)及与目标相关联的注释被视为说明性的模型组件。

(5)通用目标：通用目标之所以被称为“通用”，是因为同样的目标陈述适用于多个过程域。通用目标描述了把某一过程域相关过程制度化所必须呈现的特征，并在评估中用于确定过程域是否得到满足。通用目标的一个实例是“过程得到制度化为已定义的过程。”

(6)特定实践：特定实践是对活动的描述，该活动被认为在达成所关联的特定目标方面具有重要性。特定实践描述活动，这些活动预期带来在过程域的特定目标的达成。特定实践属

于期望的模型组件。例如，“项目监督与控制”过程域中的一条特定实践是“对照项目计划，监督所识别的承诺。”

(7)工作产品实例：工作产品实例列出了特定实践的输出例子，例如，“项目监督与控制”过程域中“监督项目计划参数”特定实践的一个工作产品实例是“重大偏差记录”。

(8)子实践：子实践是为解释和实施特定实践或通用实践提供指导的详细描述。子实践的措辞可能会让人感觉是规定的做法，不过它们实际仅旨在为过程改进提供可能有用的思路。例如，“项目监督与控制”过程域中“采取纠正措施”特定实践的一个子实践是“确定所需的适当措施并将其文档化，以处理已识别的问题。”

(9)通用实践：通用实践之所以被称为“通用”，是因为相同的实践适用于多个过程域。与通用目标相关联的通用实践描述了一些活动，这些活动被认为对通用目标的达成具有重要意义，并且有助于过程域所关联过程的制度化。例如，“过程得到制度化为已管理的过程”通用目标中的一条通用实践是“提供充分的资源，以执行过程、开发工作产品并提供过程的服务。”

(10)通用实践详细说明：通用实践详细说明出现在通用实践之后，为该通用实践在某一过程域的特定应用提供指导。例如，“项目计划”过程域中“建立并维护组织级方针，以计划并执行过程”通用实践的详细说明是“要估算计划的参数，做出内部的与外部的承诺，并且制订管理项目的计划。为此，本方针建立了组织级期望。”

下面以 CMMI 中 2 级的一个过程域“项目计划”为例，使用这个框架结构描述过程域，如图 13.6 至图 13.9 所示。

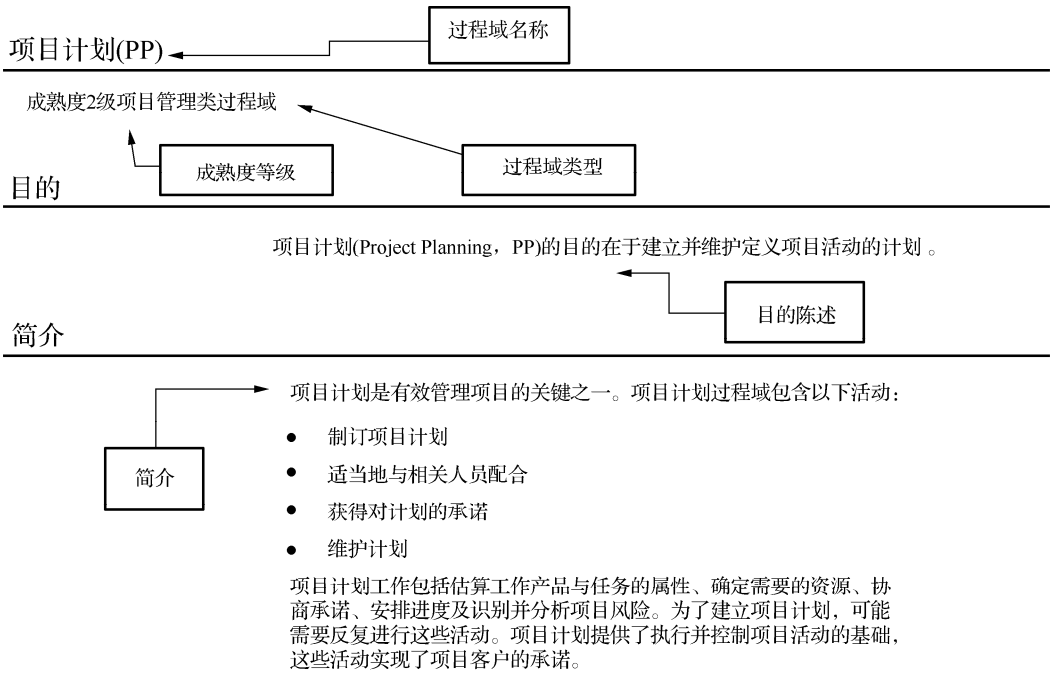


图 13.6　PP 过程域的基本描述结构

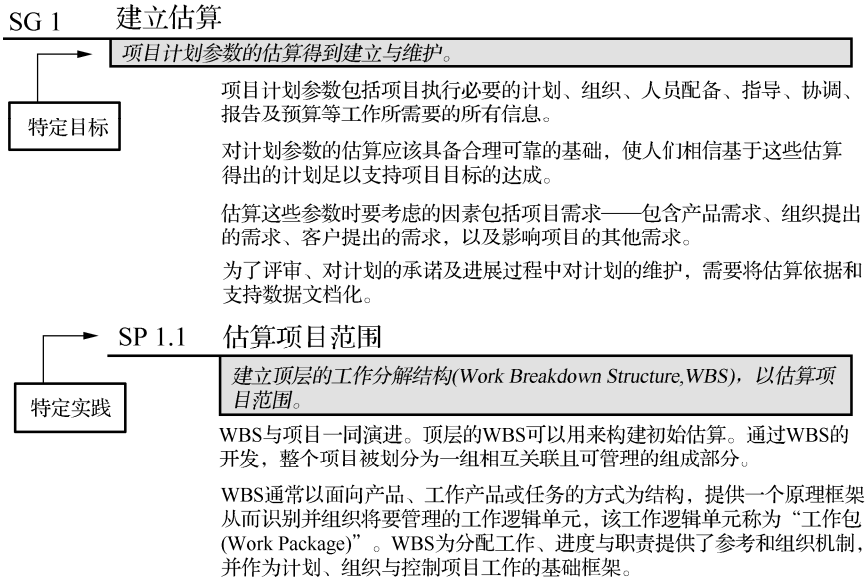


图 13.7 PP 过程域的特定目标和特定实践描述结构

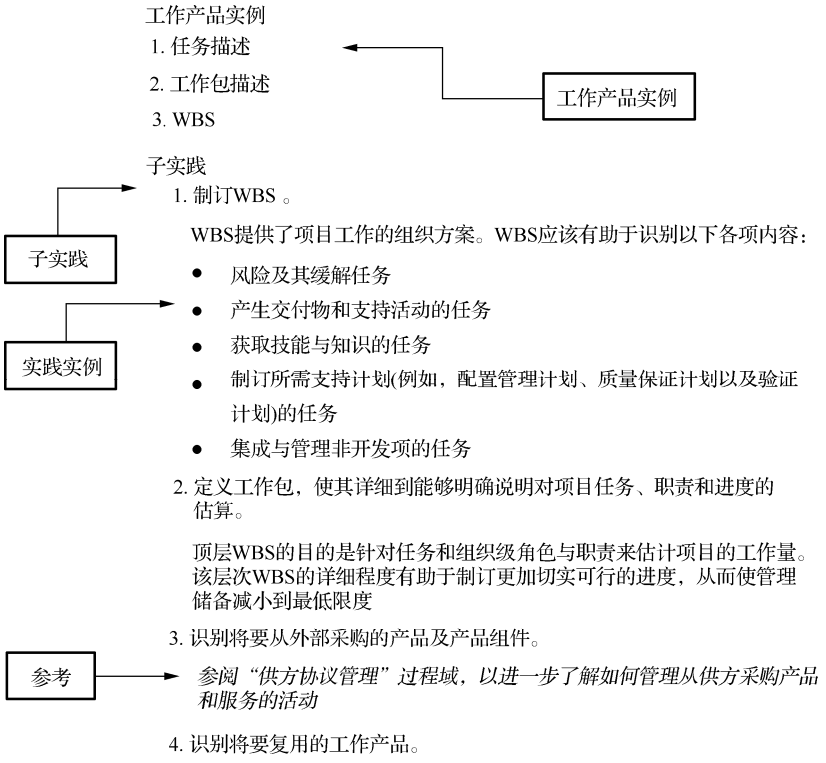


图 13.8 PP 过程域的特定子实践描述结构

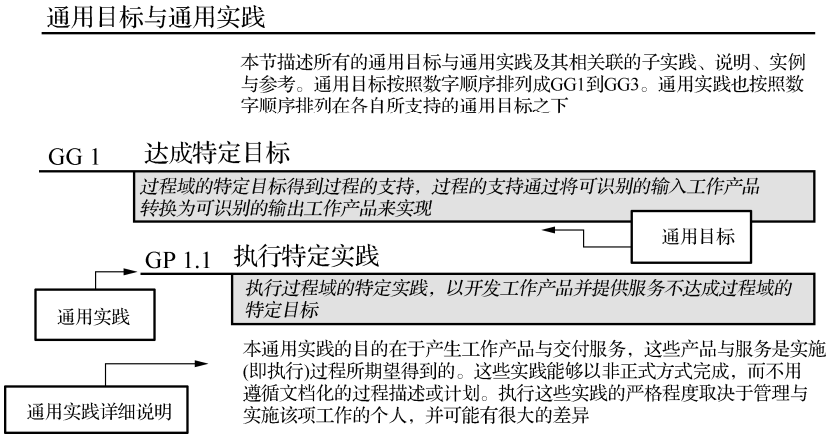


图 13.9　PP 过程域的通用目标和通用实践描述结构

在 CMMI 中，对 PP 过程域的特定目标和通用目标见表 13.2。

本书没有对 CMMI 从 2 级到 5 级的所有过程域内容做全部介绍，读者可以参考相关专业书籍和资料进一步学习。

表 13.2　PP 过程域的特定和通用目标及实践一览表

SG 1	项目计划参数的估算得到建立与维护	
	SP 1.1	建立顶层的工作分解结构(WBS)，以估算项目范围
	SP 1.2	建立和维护工作产品和任务属性的评估
	SP 1.3	定义项目生命周期阶段，以确定计划工作量的范围
	SP 1.4	根据估算原理估算项目工作产品和任务的工作量和成本
SG 2	建立和维护项目计划，作为管理项目的基础	
	SP 2.1	建立和维护项目的预算和进度
	SP 2.2	识别和分析项目风险
	SP 2.3	为项目数据的管理做计划
	SP 2.4	为执行项目的资源做计划
	SP 2.5	为项目实施所需的知识和技能做计划
	SP 2.6	为系统涉众的参与做计划
	SP 2.7	建立和维护总体项目计划
SG 3	对项目计划的承诺已确定并维护	
	SP 3.1	审查影响用于理解项目承诺的所有计划
	SP 3.2	调整项目计划，以协调可用资源和估计资源
	SP 3.3	获得相关利益获得者的承诺，负责实施和支持计划的执行
GG 2	过程被制度化为一个管理过程	
	GP 2.1	制订和维护用于策划和执行过程的组织政策
	GP 2.2	制订和维护过程执行的计划
	GP 2.3	为执行过程、开发工作产品和提供过程服务提供充足的资源
	GP 2.4	指定执行过程、开发工作产品和提供过程服务的职责和权限
	GP 2.5	根据需要培训执行或支持过程的人员
	GP 2.6	将选定的过程工作产品置于适当的控制级别下
	GP 2.7	按计划确定和涉及过程相关的系统涉众
	GP 2.8	根据过程执行的计划监控过程，并采取适当的纠正措施
	GP 2.9	客观评估过程和所选工作产品与过程描述、标准和规程的一致性，并解决不符合项
	GP 2.10	与高级管理层一起审查过程的活动、状态和结果，并且解决问题
GG 3	过程被制度化为一个已定义的过程	
	GP 3.1	建立和维护已定义过程的描述
	GP 3.2	从执行过程中收集获得的与过程相关的经验，用于支持过程的未来使用和组织过程资产的改进

13.2 个体软件过程

CMMI 框架作用的是整个组织范围，因此我们需要知道引入 CMMI 后会对每位员工带来多大的影响？CMMI 的主要开发者 Humphrey 认为，CMMI 为每位员工也定义了阶段性的能力和成熟度，即个体软件过程(Personal Software Proces，PSP)，如图 13.10 所示。每个开发者在初始阶段 PSP0 能够理解软件过程的思想并且完成相应的开发工作。为了向 PSP1 升级，员工需要总结总体工作时间，对出现的缺陷进行度量。在这一阶段，员工要具有对自己的工作进行评估的能力，如能够使用功能点方法计算工作量，对过程细节的理解在这个阶段也是有帮助的，但并不是必须的，因为对相似任务的工作量评估一般都会采用类比的方法。有了对自己工作量的评估结果，开发者可以计划本人的工作安排(个人过程的改进)，并且通过实际开发时间和缺陷的度量，确定对个人过程的改进是否带来了工作结果的改进。最后阶段的 PSP2 中，要求开发者能够树立自己的质量目标并通过自我评审评估工作效率。总之，PSP 的目标是培养开发者在尽可能准确设置的时间段中完成高质量的工作。另外，应该鼓励员工把创造性的内容融入具体的任务中，这是不断重复的软件过程所欠缺的。

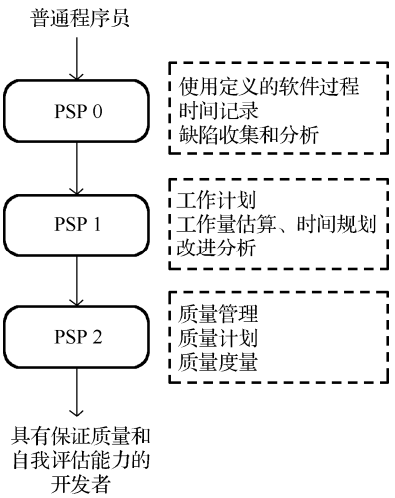


图 13.10 个体软件过程

13.3 团队软件过程

现代软件工程涉及的工作一般需要团队提供庞大和复杂的产品，这些产品不可能由一名开发者完成，开发者必须要与具备系统所需各种领域技能的人员密切合作。这需要一个工作环境，在这个环境中，人们不同的技能可以共同生产出符合其功能的高质量产品，团队软件过程(Team Software Process，TSP)技术提供了这样一个环境，它提供给团队一个框架，用来创建或裁剪团队工作流程，所有团队成员都遵循这个流程。

TSP 对软件过程的定义、度量和改进提出了一整套原则、策略和方法，把 CMMI 要求实

施的管理与 PSP 要求开发者具有的技巧结合起来，旨在按时交付高质量的软件，并把成本控制在预算的范围之内。

TSP 于 1998 产生，是建立在 PSP 之上的，目的是使团队能够更加有效地构建软件密集型产品。PSP 和 TSP 技术基于这样一个前提：一个定义的、结构化的过程可以提高个人工作质量和效率，而且需要专业人员定义过程、测量和跟踪他们的工作，帮助他们更好地理解他们所做的工作，并提供给他们必要的信息，评估和学习他们的经验。

TSP 是一个框架，在这个框架中，个人可以将自己的工作过程和技能结合团队成熟的过程管理技术，完成高质量的工作。一致和明确的过程使用也为团队协作和促进创造性提供了有效的基础和生产性工作环境。

TSP 框架结构包括一系列能力领域，如图 13.11 所示。每个能力领域都由一组相互关联的知识域组成。知识域又由概念和技能组成，这些概念和技能是知识域中包含的最小信息单位。概念用于描述 TSP 内容的智能方面，即技术的信息、事实、术语和哲学成分。技能是指个人理解和应用一个或多个概念的能力。

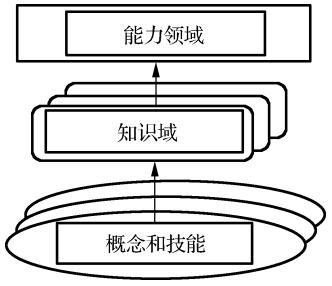


图 13.11　团队软件过程

在 TSP 中，能力领域有 6 个，分别是 TSP 基础和原理、团队基础、TSP 的项目策划、TSP 的项目实施和跟踪、收集和使用 TSP 数据、扩大 TSP。每个能力领域都全面描述了相关的知识域及知识域中的概念与技能，供团队应用到项目管理中。

当过程满足了一些预定的要求和约束，可以按照某些标准对过程进行认证。很多业务领域有不同形式的认证标准，如常见的 ISO 9000 质量体系，能够在很多不同的业务领域使用。ITIL（Information Technology Infrastructure Library）信息技术基础架构库主要适用于 IT 服务管理（ITSM），ITIL 为企业的 IT 服务管理实践提供了一个客观、严谨、可量化的标准和规范。SPICE（Software Process Improvement and Capability Determination）是一种类似于 CMMI 的机制，也提供了对软件开发公司的过程质量进行评估的框架。

最后，成功的质量管理，仅仅将优化的过程和准则记录下来不足以在实践中发挥作用，过程首先要以制度的形式体现出来，为员工所接受，只有当员工意识到过程改进涉及每个人的切身利益并且都乐意为之努力和付出时，在全员参与的基础上，才能实现全面质量管理的思想，这时的软件过程是一个企业的财富。

13.4　习题

1．CMMI 阶段式表述中 5 个等级的关注点分别是什么？
2．CMMI 中过程域 PA 分为哪些类型？每一种类型中都有哪些过程域？
3．举例说明 CMMI 的过程域 PA 的结构及其作用。
4．简单说明 PSP 的含义。
5．简单说明 TSP 的含义。